3DES

Jakub Mikuła

Opis

Aplikacja stworzona do przedstawienia działania algorytmu 3DES z wykorzystaniem konsoli windows. Aplikacja została napisana z wykorzystaniem języka Python. Do weryfikacji poprawności działania programu może zostać wykorzystana https://emvlab.org/descalc/ .

Działanie aplikacji

Aplikacja może zostać opalona za pomocą dwukrotnego kliknięcia w załączony plik exe. Jako, że aplikacja jest konsolowa – zostanie otwarta konsola w której użytkownik może wpisać tekst, na którym chce przetestować działanie algorytmu 3DES. Do zamknięcia aplikacji wystarczy kliknąć enter (bez wpisywania żadnych danych wejściowych do algorytmu).

1. Zamiana tekstu na liczby w formacie szesnastkowym
2. Zamiana liczb w formacie szesnastkowym na liczby binarne
3. Wygenerowanie 64-bitowego klucza nr.1
4. Wygenerowanie 64-bitowego klucza nr.2
5. Enkrypcja za pomocą algorytmu 3DES
6. Wypisanie otrzymanego szyfrogramu jako liczba szesnastkowa
7. Wypisanie otrzymanego szyfrogramu jako liczba binarna
8. Dekrypcja za pomocą algorytmu 3DES
9. Wypiasnie otrzymanego tekstu jako liczba binarna
10. Wypisanie otrzymanego tekstu jako liczba szesnastkowa
11. Wypisanie otrzymanego tekstu jako tekst

Kod (bez funkcji wypisujących):

```
        input_text_hex = str_to_hex(input_text_plain)
        input_text_binary = hex_to_binary(input_text_hex)

        key_1 = random_binary_string(64)
        key_1_hex = binary_to_hex(key_1)[2:]
        key_2 = random_binary_string(64)
        key_2_hex = binary_to_hex(key_2)[2:]

        encrypted_binary = triple_des_encrypt(input_text_binary, key_1, key_2)
        encrypted_hex = binary_to_hex(encrypted_binary)[2:]

        decrypted_binary = triple_des_decrypt(encrypted_binary, key_1, key_2)
        decrypted_hex = binary_to_hex(decrypted_binary)[2:]
        decrypted_plain = codecs.decode(decrypted_hex, "hex").decode()
```

Wykorzystane tabele

Initial permutation:

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Final permutation:

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Permutation:

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Expansion function:

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Permuted choice 1:

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Permuted choice 2:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

+ standardowe S-boxes

Des

Enkrypcja

Funkcja służąca do enkrypcji tekstu podanego przez użytkownika przyjmuje dwa argumenty:

- Podany przez użytkownika tekst przekonwertowany do liczb binarnych
- 64 bitowy, wygenerowany automatycznie klucz w formie binarnerj

I zwraca obiekt typu string, który jest szyfrogramem.

```
def des_encrypt(binary_text: str, binary_key: str) -> str:
```

Sprawdzana jest długość podanego tekstu w formie binarnej. Jeżeli długość zadanego tekstu nie jest podzielna przez 64 – dodawane są zera na końcu tekstu:

```
    # Manage binary_text len
    while len(binary_text)%64:
        binary_text+='0'
```

64 bitowy klucz jest zamieniany na klucz 56 bitowy (permutacja PC_1) i zostaje stworzone 16 różnych podkluczy (każdy z odpowiednim przesunięciem binarnym) dla każdej rundy algorytmu. Następnie dla każdego podklucza dokonywana jest permutacja za pomocą tabeli PC_2:

```
    # Manage key
    str_binary_key = permutation(binary_key, PC_1)
    subkeys = create_subkeys(str_binary_key)

    for index, subkey in enumerate(subkeys):
        subkeys[index] = permutation(subkey, PC_2)
```

Podany tekst jest dzielony na 64 bitowe kawałki (w przypadku zadanego tekstu o długości większej niż 64 bity):

```
# Split input text into 64 chunks
    binary_text_chunks = [binary_text[i:i+64] for i in range(0,len(binary_text),
64)]
```

Dla każdego kawałka tekstu wykonywane są poniższe czynności:

- Permutacja za pomocą tabeli Initial permutation
- Podział na lewy i prawy kawałek tekstu
- Dla każdego wygenerowanego klucza wykonywane są poniższe czynności:
    - Rozszerzenie prawej strony tekstu za pomocą Expansion function
    - Wykonanie funkcji XOR na podstawie prawej strony tekstu i aktualnego podklucza
    - Skrócenie prawej strony tekstu z 48 bitów do 32 bitów z wykorzystaniem sboxów
    - Wykoananie permutacji za pomocą tabeli Permutation
    - Wykonanie funkcji XOR na podstawie uzyskanej wartości i lewej strony tekstu
    - Lewa strona = oryginalna prawa strona tekstu
- Połączenie prawej i lewej strony tekstu
- Wykonanie ostatniej permutacji na uzyskanym tekście za pomocą tabeli Final permutation
- Zapisanie wyniku do zmiennej pomocniczej a, łączącej wyniki przejścia algorytmu dla każdego 64 bitowego tekstu.

```
    for input_text_binary_chunk in binary_text_chunks:
        input_text_binary_chunk = permutation(input_text_binary_chunk, IP)

        L = input_text_binary_chunk[:int(len(input_text_binary_chunk)/2)]
        R = input_text_binary_chunk[int(len(input_text_binary_chunk)/2):]

        for subkey in subkeys:
            tmp = R
            R = apply_Expansion(E, R)
            R = XOR(R, subkey)

            temp = ''
            for index, bits_6 in enumerate(split_in_6bits(R)):
                first_last = get_first_and_last_bit(bits_6)  # '10' -> 2
                middle4 = get_middle_four_bit(bits_6)  # '0000' -> 0
                temp += sbox_lookup(index, first_last, middle4)

            temp = permutation(temp, P)
            R = XOR(temp, L)

            L = tmp

        RL = R+L
        RL = permutation(RL, FP)
        a += RL
```

Dekrypcja

Funkcja do dekrypcji różni się jedynie kolejnością wykorzystania stworzonych podkluczy:

```
      […]
        for subkey in reversed(subkeys):
            tmp = R
            R = apply_Expansion(E, R)
            R = XOR(R, subkey)
             […]
```

3DES

Algorytm 3DES został oparty na funkcjach służących do enkrypcji i dekrypcji algorytmem DES.

Enkrypcja

Enkrypcja została zaimplementowana na podstawie wzoru:

$$C = DES_{K_1}\left(DES^{-1}_{K_2}\left(DES_{K_1}(M)\right)\right).$$

Funkcja służąca do enkrypcji:

```python
def triple_des_encrypt(plain_text: str, K1: str, K2: str) -> str:
    '''
    M - plain_text
    K1 - klucz 64 bity
    K2 - klucz 64 bity

    X = des_encrypt(M, K1)
    Y = des_decrypt(X, K2)
    encrypted = des_encrypt(Y, K1)
    '''
    X = des_encrypt(plain_text, K1)
    Y = des_decrypt(X, K2)
    encrypted = des_encrypt(Y, K1)

    return encrypted
```

Dekrypcja

Dekrypcja została zaimplementowana na postawie wzoru:

$$M = DES^{-1}_{K_1}\left(DES_{K_2}\left(DES^{-1}_{K_1}(C)\right)\right).$$

Funkcja służąca do dekrypcji:

```python
def triple_des_decrypt(ciphertext: str, K1: str, K2: str) -> str:
    '''
    M - ciphertext
    K1 - klucz 64 bity
    K2 - klucz 64 bity

    X = des_decrypt(M, K1)
    Y = des_encrypt(X, K2)
    decrypted = des_decrypt(Y, K1)
    '''
    X = des_decrypt(ciphertext, K1)
    Y = des_encrypt(X, K2)
    decrypted = des_decrypt(Y, K1)

    return decrypted
```

Wybrane funkcje

Funkcja do generowania losowych kluczy binarnych o długości 64 bitów:

```python
def random_binary_string(length):
    """ Generate a random binary string with the specified length"""
    binary_string = ''.join(str(random.randint(0, 1)) for _ in range(length))
    return binary_string
```

Funkcja do tworzenia podkluczy - create_subkeys:

```python
def create_subkeys(key_bits: str) -> list:
    left_bits, right_bits = split_in_half(key_bits)
    temp_shift_list = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]
    subkeys_to_ret = []

    for temp_shift in temp_shift_list:
        left_bits = circular_left_shift(left_bits, temp_shift)
        right_bits = circular_left_shift(right_bits, temp_shift)

        subkeys_to_ret.append(left_bits + right_bits)

    return subkeys_to_ret
```