Implementation of a symmetric key cryptographic algorithm variant.

Author - Rodrigo Lima, 98475

DETI, University of Aveiro

November 13, 2022

Contents

1 Introduction

This work consists of an implementation of a variant of the asymmetric key algorithm DES (Data Encryption Standard), named E-DES, with a key size of 256 bits and Feistel networks that are faster than the original algorithm.

As well as being implemented in two programming languages, the variant was compared to the default implementation (DES) on its computational time complexity.

DES is considered a milestone in the field of symmetric block ciphers. In spite of this, it became obsolete due to two factors: slowness and the reduced key size used. Permutations performed in software are mainly responsible for the slowness.

When the algorithm was standardized, several people considered the original key dimension, 56 bits, to be too short.

This project aims to study a variant of DES, called E-DES. In this variant, Feistel Networks and S-Boxes are the only operations used to implement a cipher similar to DES. As with many other ciphers, DES relies on S-Boxes as a building block. It employs static S-Boxes, which raises concerns about hidden cryptoanalysis trapdoors. We will use variable, key-dependent S-Boxes in E-DES and longer, 256-bit keys compared to DES.

2 Architecture

In the E-DES implementation, we are required to use an S-Box as the only function in contrast to the multiple functions used on DES.

The S-Box should be generated from the key, and it shouldn't allow the discovery of an S-Box from the value of the 15 other S-Boxes. The S-Boxes should be distinct from each other and shouldn't allow the discovery of the key that generated them.

To approach this problem, firstly the key is passed through a trapdoor function that disables the discovery of the key used from the 16 S-Boxes used:

```
void getKey(char *filePath, unsigned char* seed)
{
        ...
        // Save key
        ...
            // Generate seed from key
            SHA256(seed, strlen(seed), seed);

        ...
    }
}
```

As soon as we obtain our seed from the key, we should generate the SBoxes so they are distinct from one another and do not depend on one another. This will be implemented by generating multiple seeds based on the main seed, and these seeds will be used to generate the individual SBoxes:

```
void generate_Seeds(int seeds[16], int stringInt)
{
    for (int  i = 0 ; i < 16 ; i++){
        seeds[i] = ((i + 2)*stringInt) % 99999;
    }
}
```

SBoxes should contain bits ranging from 0 to 256 as their contents. This is achieved by starting all SBoxes with the bits needed, then rearranging the order of their contents. In order to shuffle them, the Fisher-Yates algorithm is used, and the pseudo-random number is taken from the seed generated for each SBox.

To generate pseudo-random numbers (PRNGs), the seed is first converted to an int:

```
int stringToInt(char seed[])
{
    int result = 0;

    for (int i = 0 ; i < strlen(seed) ; i++){
        result += seed[i] * (i + 1);
    }

    return result % 99999;
}
```

Now that the seed has been converted to an int, we can use it as the first int in the PRNG, and shuffle the initial SBoxes according to the algorithm below:

```
void shuffle(uint8_t *array, int seed)
{
    int i, j;
    uint8_t tmp;

    for (i = 255 ; i > 0; i--) {
        seed = getNextInt(seed);
        j = seed % 256;
        tmp = array[j];
        array[j] = array[i];
        array[i] = tmp;
    }
}
```

A nonlinear-feedback shift register is used to generate pseudo-random numbers. This NFSR is based on the algorithm presented in Jean-Philippe Aumasson's book "Serious Cryptography A Practical Introduction to Modern Encryption". As an alternative to bits, the implementation uses decimal digits as follows:

```
//Non-liniear FSRs
static int getNextInt(int seed)
{
    int n1 = (int) (seed / 10) % 10;
    int n2 = (int) (seed / 100) % 10;
    int n3 = (int) (seed / 1000) % 10;
    int n4 = (int) (seed / 10000) % 10;
    int n5 = (int) (seed / 100000) % 10;

    return ((n1*n3*n4 + n1*n2 + n2*n3 + n3*n4 + n1 + n2) * seed + n5) % 99999;
}
```

Thus, we will be able to generate SBoxes with the necessary requirements.

3 Conclusion

In each of the two languages where E-DES variants were implemented, 100000 encryptions and decryptions of a file with 4 kilobytes of random data from /dev/urandom were performed and the fastest times were recorded.

The following times were obtained for the C language:

ALgorithm used	Action made	Seconds
E-DES	Encryption	0.000109
E-DES	Decryption	0.000111
DES	Encryption	0.000226
DES	Decryption	0.000228

The following times were obtained for the Go language:

ALgorithm used	Action made	Seconds
E-DES	Encryption	0.008694
E-DES	Decryption	0.005131
DES	Encryption	0.002090
DES	Decryption	0.001399

According to the results, the C implementation of E-DES takes, in the best scenario, half as long as the DES implementation. This is predictable, the DES implementation implements 16 Feistel rounds in addition to two permutations made at the beginning and at the end.

The same cannot be seen on the Go implementation since no Goroutines were used in the performance of E-DES, which made decoding more time-consuming than the DES implementation from the crypto library.