HW2 - KEY RECOVERY PROGRAM

In this second homework of the year, we were asked to create a program capable of recovering the encryption keys from a given plaintext-ciphertext pair. The encryption algorithm has been created for this homework and combines DES and AES.
This program had to be done in C language only.

PART 1 - Algorithm

The algorithm for this assignment is based on the Meet In The Middle principle seen in class. This method allows to exploit the structure of multi-encrypted schemes by breaking a huge problem into two smaller ones. Instead of bruteforcing all the possible keys, giving a complexity of O(N**2), the attack by Meet In The Middle only has a complexity of O(2N), which is a very significant improvement, especially considering that more than 180,000 passwords were presented in the given TXT file. This reduces the amount of operations from more than 32Million to 360,000 only. It works simply because encrypting twice a text with two different keys is vulnerable. Indeed, because there is an intermediate value, it is possible to “brute force” our way from both sides of the algorithm and simply compare the intermediate value to find the two keys at once. This means that the attack is composed of two phases, the forward and the backward attack. The first step (forward attack), consists of going through all of the possible passwords, and encrypt the plaintext using DES. These values are then stored as “intermediate” in a hash table. The second phase (backward attack) decrypts the cipher text using all the potential keys, and compares the result with the firstly found results. If a match is found, then the keys are found also.

PART 2 - Code

My code is separated in 10 different functions:

1) readPasswordFile()
2) CreateHashTable()
3) getCP()
4) hashFunction()
5) InsertHashTable()
6) SearchHashTable()
7) base64_decode()
8) des_ecb_encrypt()
9) aes_cbc_decrypt()
10) main()

The **readPasswordFile** is used to read the password.txt file that was provided for the assignment. It is of type “password”, a newly created structure that stores both the MD5 and the password itself. It returns a list of password structures.

The **createHashTable** function is simply responsible for creating the hash table that will be used to store and go through all of the intermediate values. It returns an item of type HashTable, again a new structure that simply contains his size and the head of the hash list.

The **getCP** function is used to retrieve from the input the ciphered text and the plain text and to store them in different variables, passing them as a structure called CP, that contains both the cipher and plain texts. It is not necessary to use this technique, but I did it early in the project and didn’t want to remove it.

The **hashFunction** function is responsible for hashing the data before passing them in the hash table. It is used in the two following functions. This function uses the djb2 algorithm. This algorithm was recommended for its implementation simplicity and speed.

The **InsertHashTable** function is used to efficiently insert new items in the hash table. It uses the **hashFunction** seen beforehand to store the values as hashed, and creates a new item of Hash type, which is a structure that contains the key, its size, the password and the following hash. It is mainly used in the forward attack.

The **SearchHashTable** function is used to search for specific keys in the hash table by going through the linked list. It also hashes the input before searching for the key. It is mainly used for the backward attack.

The **base64_decode**, **des_ecb_encrypt** and **aes_cbc_decrypt** functions are used for encryption and decryption using different techniques, base64 to decode the cipher text in the beginning, the des encryption is for the forward attack, and the aes decryption for the backward attack.

Finally, the **main** function is linking all the functions together, but it is also responsible for padding the plaintext, storing the results in a file, as well as freeing the memory.

PART 3 - Performance Analysis

This algorithm's efficiency and performance depends on the amount of possible passwords, as well as their size. It is efficient in the context of this assignment because of the known password list and fixed sizes. It would be less efficient in real case scenario.
For N the number of passwords and M the size of the plaintext, the complexity is as followed:

Step	**Time Complexity**	**Space Complexity**
Read & hash passwords	O(N)	O(N)
Forward DES attack	O(N * M)	O(N * M)
Backward AES attack	O(N*M)	O(1)
Hash table lookup	O(1)	O(N*M)

The overall complexities of the algorithm are the following:

- **Time Complexity :** O(N * M)
- **Space Complexity :** O(N * M)

We can conclude that this algorithm is efficient in the scope of this assignment, but would be less in a real case scenario with less information accessible.