Secure Processor MicroArchitecture
Assignment 1

Team Members:
Surya Prasad S - EE19B121
Sai Dheeraj Ettamsetty - CS18B055

Question 1

We have used the CLEFIA reference Implementation given along with the assignment.

We have used this specification document to understand how to Implement the T-table Implementation of CLEFIA. The implementation we have followed is in Page.46 of this specification document linked above.

Implementation Details:

The main purpose of using T-tables is to avoid looking into S-boxes and also avoid Multiplication with Diffusion Matrix. The computations are done beforehand for all possible 8-bit inputs and are stored in the respective T-tables. As we will have 256 different strings with 8-bits, the size of each of these T-tables is 256. We have used `unsigned long` arrays to store the T-tables.

$$
\begin{aligned}
T_{00}(x) &= (\ S_0(x), & \{02\} \times S_0(x), & \{04\} \times S_0(x), & \{06\} \times S_0(x)) \\
T_{01}(x) &= (\{02\} \times S_1(x), & S_1(x), & \{06\} \times S_1(x), & \{04\} \times S_1(x)) \\
T_{02}(x) &= (\{04\} \times S_0(x), & \{06\} \times S_0(x), & S_0(x), & \{02\} \times S_0(x)) \\
T_{03}(x) &= (\{06\} \times S_1(x), & \{04\} \times S_1(x), & \{02\} \times S_1(x), & S_1(x))
\end{aligned}
$$

T-tables for the function ClefiaF0Xor

$$
\begin{aligned}
T_{10}(x) &= (\ S_1(x), & \{08\} \times S_1(x), & \{02\} \times S_1(x), & \{0A\} \times S_1(x)) \\
T_{11}(x) &= (\{08\} \times S_0(x), & S_0(x), & \{0A\} \times S_0(x), & \{02\} \times S_0(x)) \\
T_{12}(x) &= (\{02\} \times S_1(x), & \{0A\} \times S_1(x), & S_1(x), & \{08\} \times S_1(x)) \\
T_{13}(x) &= (\{0A\} \times S_0(x), & \{02\} \times S_0(x), & \{08\} \times S_0(x), & S_0(x))
\end{aligned}
$$

T-tables for the function ClefiaF1Xor

Each of these 8 T-tables take 8-bit input and give 32-bit output. The C program to generate these 8 T-tables in a single run is also provided in our source folder with the name, ‘get_t_tables.c’.
We copied the T-tables into our C code which is a modified version of ‘clefia_ref.c’.
Two changes are made targeting two functions, ClefiaF0Xor function and other is ClefiaF1Xor function. Care is required while copying the ‘long’ values of T-tables into a character array of size 4 as it is expected by the next block of code. Finally, all the function definitions have been added to ‘clefia_t_table.h’ and included in main.c.

```
(base) saidheeraj@saidheeraj:~/SPM-2022/Assgn1/CLEFIA$ make
gcc main.c clefia_t_table.c -o main
./main
--- Test ---
plaintext:  000102030405060708090a0b0c0d0e0f
secretkey:  ffeeddccbbaa99887766554433221100f0e0d0c0b0a0908070605040302010 00
--- CLEFIA-128 ---
ciphertext: de2bf2fd9b74aacdf1298555459494fd
plaintext : 000102030405060708090a0b0c0d0e0f
(base) saidheeraj@saidheeraj:~/SPM-2022/Assgn1/CLEFIA$
```

Question 2

a) The stored password is **spm{fastandfurious}**

b) The stored password is of length 19. We cannot have another string of length 19 which has the same hash as our password string. This is because of the way we are generating the hashes, wherein each character is substituted by another unique character to form the hash. One thing we noticed was that the hash function ignores characters which are present after the 19th position. So we can have strings that are more than 19 characters long which result in the same hash as that of the stored password.
The following are 5 such strings:
1. **spm{fastandfurious}a**
2. **spm{fastandfurious}b**
3. **spm{fastandfurious}c**
4. **spm{fastandfurious}d**
5. **spm{fastandfurious}e**

c) We haven't used any references for creating our code. The approach we used is very simple and is as follows:

i) We start with a list of all possible characters and initialize our password guess (pwd_guess) randomly with the characters from the list. We used randint() because it is possible for repetition of characters.

```
s = "abcdefghijklmnopqrstuvwxyz{}=_"
given_chars = list(s)

pwd_guess = ''
for i in range(prime):
   pwd_guess = pwd_guess + given_chars[random.randint(0, 30)]
```

ii) Two nested for loops are needed, one to iterate over each position and the other to test all the characters from given_chars. In each iteration of the outer for loop,

we randomly shuffle the list of given characters to increase the probability of getting a hit. The logic we are using is that if the targeted character is identified the time taken will be more than the time taken when it is misidentified (as more characters are tested). The amount of time taken for misidentification is approximately equal to i+1 for the given password checker function. One issue with this logic is that for the last character the time taken cannot reveal whether it was a success or not. Hence, we need the pwd_found flag to be set if Access is Granted.

```
# A flag which stores if we found the password.
pwd_found = False

for i in range(prime):  # 19 times
   random.shuffle(given_chars)
   # Pick a position
   pos = (7*i+4)%prime
   print("Attempting identification of position no.", pos)

   for j in range(len(given_chars)):   # max 30 times
       bashcmd = f"echo \"{pwd_guess}\" | nc 10.21.235.179 5555 >
output.txt"

       os.system(bashcmd)
       # output.txt will have the output given by the binary
       with open('output.txt', 'r') as f:
           for line in f:
               if "Time taken to verify" in line:
                   runtime = int(float(line.split()[-1]))

               if "Access Granted" in line:
                   print("Password identified!")
                   pwd_found = True

           if pwd_found:
               break
           if runtime == i+1:    # character at pos did not match
               pwd_guess = pwd_guess[:pos] + given_chars[j] +
pwd_guess[pos+1:]
           else: # char at pos matched, break and pick another char
               break
```

```
    if pwd_found:
        break

print("Password is", pwd_guess)
```

iii) The position targeted in each iteration is also determined in the initial part of the block of code. It is not in the continuous order of pwd_guess as the hash output is only tested continuously and that processes the input password in a different way.

iv) As it can be seen in the above code, we use bash commands and invoke the OS to execute them. This allows us to dynamically decide and execute commands as required for the timing attack. Linux offers support piping commands so that we don't have to directly interact as the user.

v) The inner for loop is expected to encounter the break statement every time a correct character is identified. All the for loops are halted when "Access Granted" message is received.

```
Current guess: mtc=szaognhe{}bdyup
Attempting identification of position no. 4
Current guess: mtc=fzaognhe{}bdyup
Attempting identification of position no. 11
Current guess: mtc=fzaognhf{}bdyup
Attempting identification of position no. 18
Current guess: mtc=fzaognhf{}bdyu}
Attempting identification of position no. 6
Current guess: mtc=fzsognhf{}bdyu}
Attempting identification of position no. 13
Current guess: mtc=fzsognhf{rbdyu}
Attempting identification of position no. 1
Current guess: mpc=fzsognhf{rbdyu}
Attempting identification of position no. 8
Current guess: mpc=fzsoanhf{rbdyu}
Attempting identification of position no. 15
Current guess: mpc=fzsoanhf{rboyu}
Attempting identification of position no. 3
Current guess: mpc{fzsoanhf{rboyu}
Attempting identification of position no. 10
Current guess: mpc{fzsoandf{rboyu}
Attempting identification of position no. 17
Current guess: mpc{fzsoandf{rboys}
```

```
Attempting identification of position no. 5
Current guess: mpc{fasoandf{rboys}
Attempting identification of position no. 12
Current guess: mpc{fasoandfurboys}
Attempting identification of position no. 0
Current guess: spc{fasoandfurboys}
Attempting identification of position no. 7
Current guess: spc{fastandfurboys}
Attempting identification of position no. 14
Current guess: spc{fastandfurioys}
Attempting identification of position no. 2
Current guess: spm{fastandfurioys}
Attempting identification of position no. 9
Current guess: spm{fastandfurioys}
Attempting identification of position no. 16
Password identified!
Password is spm{fastandfurious}
```

vi) The whole process shown above took us 3294 seconds! One minor improvement which can be done in our code is if we consider multiple hits in each iteration but its significance is very small.

vii) After 19 iterations, our password guess string `pwd_string` will contain all the correct characters.

```
(base) saidheeraj@saidheeraj:~/SPM-2022/Assgn1$ nc 10.21.235.179 5555
Enter the password:
spm{fastandfurious}
Access Granted
Time taken to verify =  19.020558078002068

(base) saidheeraj@saidheeraj:~/SPM-2022/Assgn1$
```

THE END