CS6630: Secure Processor Microarchitecture
Assignment 1: Cryptography 101

EE19B026
J. Phani Jayanth

Question 1: CLEFIA-128

About CLEFIA

CLEFIA is a *block cipher* algorithm, developed by SONY, which was first published in 2007. It is an efficient and highly secure block cipher with a *block length* of 128 bits, while *key lengths* of 128 bits, 192 bits, and 256 bits are possible. It was intended to be used in DRM Systems. CLEFIA provides advanced capabilities in wide-ranging environments, even in restrictive environments such as smart cards and mobile devices. CLEFIA is internationally standardized in ISO/IEC 29192-2:2019 Lightweight Cryptography. (Reference)

CLEFIA Block Cipher Implementation

The block cipher algorithm is divided into two parts - *data processing* part and *key scheduling* part. CLEFIA employs a generalized *Feistel* structure with 4-Branch Data Lines with two *F-functions* in a round, located in parallel, and their input/output length is 32 bits. The 4-branch structure enables compact implementation of CLEFIA in hardware and software. CLEFIA utilizes Diffusion Switching Mechanism (DSM) for immunity against both differential and linear attacks with fewer rounds, making the block cipher faster.

Two different S-boxes are used in CLEFIA (F_0 & F_1) - An 8-bit S-box based on 4-bit S-boxes, and another 8-bit S-box based on an inversion in Galois Field GF(2^8). A simple linear operation *DoubleSwap* is used in the *key scheduling* part, which splits the 128-bit input into 4 parts and shuffles them. *DoubleSwap* can be used for rapidly generating round keys while maintaining hardware and software implementation efficiency. (Reference)

T-Table Implementation

T-Tables are a way to optimize the implementation of a cipher algorithm by relying on pre-computed lookup tables. This kind of optimization was originally proposed for the AES Algorithm. In CLEFIA, the *Substitution* layer and *Diffusion* layer computations can be supplanted with lookup tables (T-Table).

As mentioned earlier, F_0 & F_1 S-boxes are used in CLEFIA. The substitution happens as follows:

$$\mathbf{T_0 \leftarrow S_0[X_0]}$$
$$\mathbf{T_1 \leftarrow S_1[X_1]}$$
$$\mathbf{T_2 \leftarrow S_0[X_2]}$$
$$\mathbf{T_3 \leftarrow S_1[X_3]}$$

Diffusion layer matrices (or computations) are different in F_0 & F_1.
For $\boldsymbol{F_0}$:

$$\mathbf{Y_0 = T_0 \oplus 2*T_1 \oplus 4*T_2 \oplus 6*T_3}$$
$$\mathbf{Y_1 = 2*T_0 \oplus T_1 \oplus 6*T_2 \oplus 4*T_3}$$
$$\mathbf{Y_2 = 4*T_0 \oplus 6*T_1 \oplus T_2 \oplus 2*T_3}$$
$$\mathbf{Y_3 = 6*T_0 \oplus 4*T_1 \oplus 2*T_2 \oplus T_3}$$

For $\boldsymbol{F_1}$:

$$\mathbf{Y_0 = T_0 \oplus 8*T_1 \oplus 2*T_2 \oplus 10*T_3}$$
$$\mathbf{Y_1 = 8*T_0 \oplus T_1 \oplus 10*T_2 \oplus 2*T_3}$$
$$\mathbf{Y_2 = 2*T_0 \oplus 10*T_1 \oplus T_2 \oplus 8*T_3}$$
$$\mathbf{Y_3 = 10*T_0 \oplus 2*T_1 \oplus 8*T_2 \oplus T_3}$$

where **(*)** denotes multiplication in Galois Field GF(2^8) with $p(x) = x^8 + x^4 + x^3 + x^2 + 1$

Now, the transformation from $\mathbf{X\{X_0,X_1,X_2,X_3\} \rightarrow Y\{Y_0,Y_1,Y_2,Y_3\}}$ can be captured. Considering the case for $\boldsymbol{F_0}$:

$$\mathbf{Y_0 = S_0[X_0] \oplus 2*S_1[X_1] \oplus 4*S_0[X_2] \oplus 6*S_1[X_3]}$$
$$\mathbf{Y_1 = 2*S_0[X_0] \oplus S_1[X_1] \oplus 6*S_0[X_2] \oplus 4*S_1[X_3]}$$
$$\mathbf{Y_2 = 4*S_0[X_0] \oplus 6*S_1[X_1] \oplus S_0[X_2] \oplus 2*S_1[X_3]}$$
$$\mathbf{Y_3 = 6*S_0[X_0] \oplus 4*S_1[X_1] \oplus 2*S_0[X_2] \oplus S_1[X_3]}$$

$$\mathbf{Y = Y_0 \mid Y_1 \mid Y_2 \mid Y_3}$$

Here **(|)** represents a *concatenation.*

$$\mathbf{Y = S_0[X_0] \oplus 2*S_1[X_1] \oplus 4*S_0[X_2] \oplus 6*S_1[X_3] \mid 2*S_0[X_0] \oplus S_1[X_1] \oplus 6*S_0[X_2] \oplus 4*S_1[X_3] \mid}$$
$$\mathbf{4*S_0[X_0] \oplus 6*S_1[X_1] \oplus S_0[X_2] \oplus 2*S_1[X_3] \mid 6*S_0[X_0] \oplus 4*S_1[X_1] \oplus 2*S_0[X_2] \oplus S_1[X_3]}$$

Suitably re-ordering the above expression gives us the following:

$$\mathbf{Y = \{S_0[X_0] \mid 2*S_0[X_0] \mid 4*S_0[X_0] \mid 6*S_0[X_0]\} \oplus \{2*S_1[X_1] \mid S_1[X_1] \mid 6*S_1[X_1] \mid 4*S_1[X_1]\} \oplus}$$
$$\mathbf{\{4*S_0[X_2] \mid 6*S_0[X_2] \mid S_0[X_2] \mid 2*S_0[X_2]\} \oplus \{6*S_1[X_3] \mid 4*S_1[X_3] \mid 2*S_1[X_3] \mid S_1[X_3]\}}$$

The above expression is the essence of the T-Table optimization. The T-Tables are now computed as follows:

$$\mathbf{TT_0[i] \leftarrow S_0[i] \mid 2*S_0[i] \mid 4*S_0[i] \mid 6*S_0[i]}$$
$$\mathbf{TT_1[i] \leftarrow 2*S_1[i] \mid S_1[i] \mid 6*S_1[i] \mid 4*S_1[i]}$$
$$\mathbf{TT_2[i] \leftarrow 4*S_0[i] \mid 6*S_0[i] \mid S_0[i] \mid 2*S_0[i]}$$
$$\mathbf{TT_3[i] \leftarrow 6*S_1[i] \mid 4*S_1[i] \mid 2*S_1[i] \mid S_1[i]}$$

$$\mathbf{Y = TT_0[X_0] \oplus TT_1[X_1] \oplus TT_2[X_2] \oplus TT_3[X_3]}$$

Similarly, the T-Tables for F_1 can also be calculated:

$$\mathbf{TT_0[i] \leftarrow S_1[i] \mid 8*S_1[i] \mid 2*S_1[i] \mid 10*S_1[i]}$$
$$\mathbf{TT_1[i] \leftarrow 8*S_0[i] \mid S_0[i] \mid 10*S_0[i] \mid 2*S_0[i]}$$
$$\mathbf{TT_2[i] \leftarrow 2*S_1[i] \mid 10*S_1[i] \mid S_1[i] \mid 8*S_1[i]}$$
$$\mathbf{TT_3[i] \leftarrow 10*S_0[i] \mid 2*S_0[i] \mid 8*S_0[i] \mid S_0[i]}$$

In this way, **8 T-Tables** will be employed in CLEFIA.

Directory Structure

Assignment-1/CLEFIA-128

- CLEFIA.c
- CLEFIA.h
- GenTTable.c
- Makefile
- TEST_CLEFIA.c

The file *GenTTable.c* generates the necessary T-Tables for the T-Table based implementation of CLEFIA Blockcipher algorithm. The script writes the T-Tables to a *header* file - *TTable.h*.

```
$ gcc GenTTable.c -o GenTTable
$ ./GenTTable
```

The *Makefile* compiles the project and prints out the *encryption* and *decryption* data on the terminal, and generates the executable *TEST_CLEFIA*.

```
$ make
$ ./TEST_CLEFIA
```

Output

```
--- Test ---
Plain Text:  000102030405060708090a0b0c0d0e0f
Secret Key:  ffeeddccbbaa99887766554433221100
--- CLEFIA-128 ---
Cipher Text: de2bf2fd9b74aacdf1298555459494fd
Plain Text : 000102030405060708090a0b0c0d0e0f
```

Question 2: Password Extraction

(a) The Stored Password: spm{Fast_&_Furious}

The stored password is extracted by employing a *Python* script (details in section **(c)**).

```
phoenix@DESKTOP-9PO42US:/mnt/c/Users/jayan/
ion$ nc 10.21.235.179 5555
Enter the password:
spm{Fast_&_Furious}
Access Granted
Time taken to verify =  19.02050316601526
```

(b) Hash Collisions:

There is a possibility of different passwords having the same hash - ***Hash Collision***. In our case, if we consider the *hash function* implemented, each character is uniquely mapped to another different character without any correlation or correspondence. Therefore, two different characters **cannot** have the same hash transformation.

However, the *hash function* is limited to a length of **19** characters, i.e., a 19-digit password can be appended with more digits, making it greater than 19 in length, and it would have the same hash. For instance, **spm{Fast_&_Furious}** and **spm{Fast_&Furious}xyz** will have the same hash.

```
Hash of spm{Fast_&_Furious}:  FF_ute{x={uzduomb&d
Hash of spm{Fast_&_Furious}xyz:  FF_ute{x={uzduomb&d
```

Similar examples include: **spm{Fast_&_Furious}SPM, spm{Fast_&_Furious}123, spm{Fast_&_Furious}&%@, spm{Fast_&_Furious}A$7, etc.**

(c) Method of Password Extraction:

From the reference provided, we can observe how the *hash function* operates on the input password. The *hash length* is set to 19. If the provided *password* is less than 19-digits in length, the *hash function* extends it to a length of 19 by appending "=" to the passwords' tail, as shown.

```
def compute_hash(password):
    password = password.ljust(prime,"=")[0:prime]
    hash = ""
    for i in range(prime):
        hash += substitution(password[(7*i+4)%prime])
    return hash
```

A unique mapping for *substitution* is implemented, where every digit in the provided *password* is mapped to a different digit without any overlapping scenarios.

The indices are also not mapped directly, but are mapped through a transformation:

$$\mathbf{i \leftarrow (7i + 4)\%19}$$

For example, the 5th element (i = 4) of the password is mapped to the 1st element (i = 0) of the hash. Similarly, the 11th element (i = 10) of the password maps to the 2nd element (i = 1) of the hash, and so on.

The following snippet is the crux of the *Python* script implemented to extract the password:

```
for i in range(prime):
    TIMES = []
    PWD = TRYPWD.copy()
    for C in ALL_CHARS:
        PWD[HASH_IDXS[i]] = C
        PWDINP = "".join(PWD)+"\n"
        C = subprocess.Popen(["nc","10.21.235.179","5555"], stdin=subprocess.PIPE, stdout=subprocess.PIPE)
        stdout, stderr = C.communicate(input = PWDINP.encode())
        RESPONSE = stdout.decode()
        TIMES.append(float(((RESPONSE.split("\n")[2]).split("=")[1]).strip()))
        if (i==18 and (RESPONSE.split("\n")[1].strip()) == "Access Granted"):
            break;
    if (i!=18):
        TRYPWD[HASH_IDXS[i]] = ALL_CHARS[TIMES.index(max(TIMES))]
    print(TRYPWD)
```

Figure 1: *For-Loop* for extracting the stored password

```
HASH_IDXS = []
for i in range(prime):
    HASH_IDXS.append((7*i+4)%prime)
```

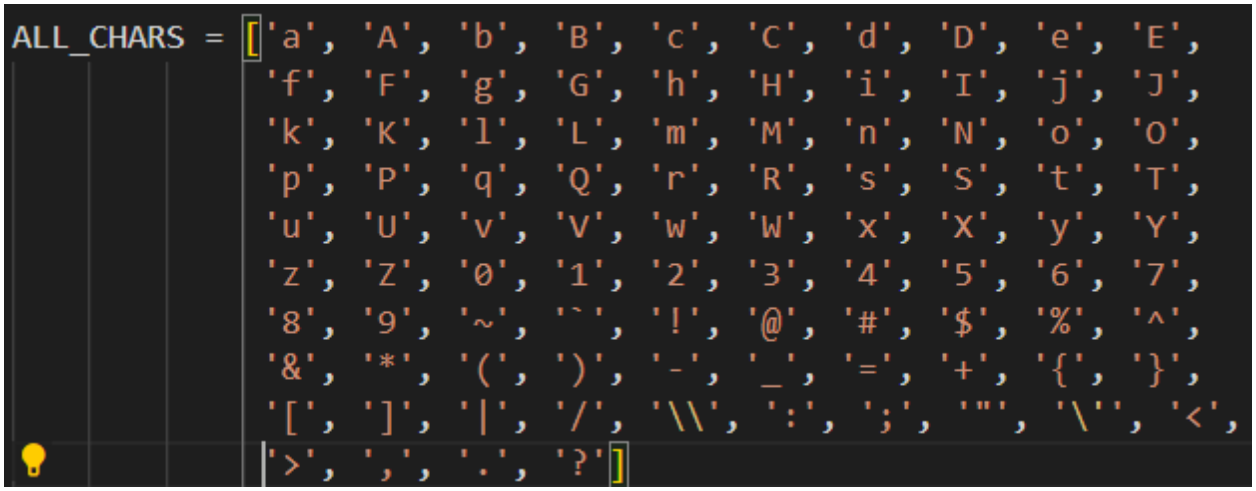
A *list* HASH_IDXS is created, which captures the index mapping between the *password* and the *hash.* This is used to designate the order in which the *stored password* is decrypted.

The *password checker* function was implemented as shown below:

```
def password_checker(password):
    hash = compute_hash(password)
    for i in range(len(flag)):
        time.sleep(1)
        if(flag_hash[i]!=hash[i]):
            return False
    return True
```

The *hash* of the provided *password* and that of the *actual password* are compared to find the correctness of the entered password. As mentioned earlier, the *1st* index of the *hash* corresponds to the *4th* index of the *password.* Therefore, while decrypting the *password*, we start by first finding out the *4th* index, then the *11th* index, and so on. This is implemented in the *for-loop* shown in Figure 1.

A *list* with all possible digits the *password* can contain is created - a *list* of digits from a-z, A-Z, 0-9, and special characters like @!#%$*&. In the *for-loop,* each digit to be decrypted is looped through all the possible digits.



Utilizing the *subprocess module* in Python, the *binary* is called and the *guessed password* is provided as the input. The subprocess then issues a *response* stating the correctness of the password, and it also provides the time taken in computing the input passwords' validity. These times are suitably appended into another *list* for finding the required digit.

The logic is as follows. As mentioned before, while *looping* through each of the possible digits, the time taken for checking the passwords' correctness is obtained from the subprocess *response.* When the correct digit is in place, the *response* takes a longer time to be issued because the *password checker* loop proceeds to the next iteration. Therefore, by collecting all such times taken by all possible digits and finding the maximum element in the *list*, we can identify the digit that must be located in the corresponding index. This is repeated for the entire length of the *hash.*

However, for the last index of the *hash,* the time taken by the *password checker* function will remain the same for the correct as well as the incorrect digits. This is because there is no next iteration to proceed to. Hence, to identify the last correct digit, we require the *"Access Granted" response* from the *password checker subprocess,* implemented as shown in Figure 1.

****