

Assignment 1

CS6630: Secure Processor Microarchitecture

Arjun Menon V, Akilesh Kannan

EE18B104, EE18B122



1 CLEFIA

The CLEFIA-128 cipher implemented here is adapted from the reference implementation provided by SONY Corporation from here.

1.1 Approach

The F_0 and F_1 functions in CLEFIA have been implemented using lookup tables (T-Tables). These functions formed the core of CLEFIA, with a *substitution* layer and a *diffusion* layer.

However, due to the diffusion layer, which involves modular arithmetic in $GF(2^8)$, timing attacks can be done, due to the if-else structure of the multiplication operation in the reference implementation.

To overcome this, we can use lookup tables to perform the F-functions (as well as the following XOR operation), which will occur in constant time¹.

We have used 8 lookup tables (8KB each) totally to implement the F_0 and F_1 functions. This can be reduced to 4 tables, with the other 4 being just a circularly shifted version of these, but we chose not to for the sake of easier understanding.

F_0 , F_1 functions

```
static void f0(uint8 *dst, const uint8 *src, const uint8 *rk)
{
    uint8 x[4];
    uint32 y;

    /* Key addition */
    byteXor(x, src, rk, 4);

    /* Substitution layer, Diffusion layer (M0) */
    y = TF00[x[0]] ^ TF01[x[1]] ^ TF02[x[2]] ^ TF03[x[3]];

    y = __builtin_bswap32(y);

    /* Xoring after F0 */
    byteCpy(dst + 0, src + 0, 4);
    byteXor(dst + 4, src + 4, (const uint8 *)y, 4);
}

static void f1(uint8 *dst, const uint8 *src, const uint8 *rk)
{
    uint8 x[4];
    uint32 y;
```

¹ Assuming, the table is fully located in cache.

```

/* Key addition */
byteXor(x, src, rk, 4);

/* Substitution layer, Diffusion layer (M1) */
y = TF10[x[0]] ^ TF11[x[1]] ^ TF12[x[2]] ^ TF13[x[3]];

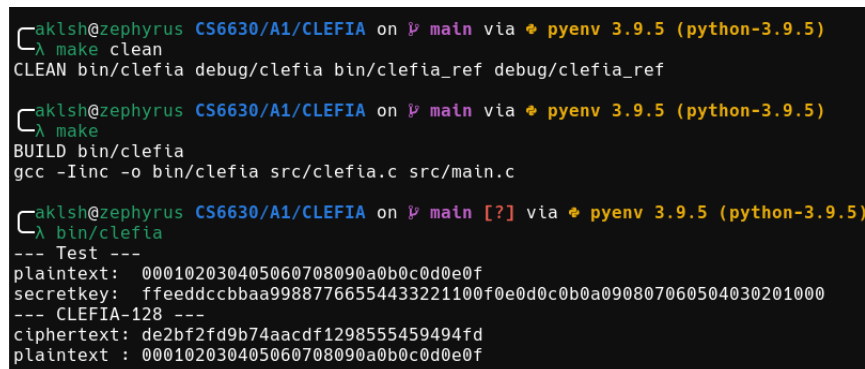
y = __builtin_bswap32(y);

/* Xoring after F1 */
byteCpy(dst + 0, src + 0, 4);
byteXor(dst + 4, src + 4, (const uint8 *)&y, 4);
}

```

In the above code snippet, the TF** are the lookup tables - the first digit denotes which of F0 or F1 it is used in and the second denotes which byte of the output it is used for. We perform an endian-ness switch² in order to make it compatible with the reference implementations' XOR function.

1.2 Results



```

aklsh@zephyrus CS6630/A1/CLEFIA on 1 main via 1 pyenv 3.9.5 (python-3.9.5)
λ make clean
CLEAN bin/clefi debug/clefi bin/clefi_ref debug/clefi_ref

aklsh@zephyrus CS6630/A1/CLEFIA on 1 main via 1 pyenv 3.9.5 (python-3.9.5)
λ make
BUILD bin/clefi
gcc -Iinc -o bin/clefi src/clefi.c src/main.c

aklsh@zephyrus CS6630/A1/CLEFIA on 1 main [?] via 1 pyenv 3.9.5 (python-3.9.5)
λ bin/clefi
--- Test ---
plaintext: 000102030405060708090a0b0c0d0e0f
secretkey: ffeeddcbbbaa99887766554433221100f0e0d0c0b0a090807060504030201000
--- CLEFIA-128 ---
ciphertext: de2bf2fd9b74aacdf1298555459494fd
plaintext : 000102030405060708090a0b0c0d0e0f

```

Fig. 1: CLEFIA-128 Cipher Output

2 Hashed Password Cracking

2.1 Approach

The following observations were made about the hash implementation:

- The hash only used the first PRIME characters of the input password. The rest of the characters didn't matter. This allows one to perform collision attacks on the hash, if the secret key was more than PRIME characters long - all the attacker has to do is to find the first PRIME characters in the secret key.
- Each character in the hashed string was dependent only on a single character from the input string. This allows one to easily reverse-engineer the hash by sequentially brute-forcing each of the characters in the correct order as they were used by the hashing mechanism. This reduced the number of tries exponentially - from PRIME^N to $\text{PRIME} * N$, where N is the number of characters in the search space.

² This is not required in a big-endian machine. Code was tested only on little-endian machines.

Based on these observations, we wrote a python script that starts with a known password (the blank password), and sequentially try brute-forcing the characters in the correct order as used by the hash. The correct character can be identified by a spike in the time taken for the hash computation (will increase by 1 sec). The next iteration of the cracker will use this character and try guessing the next character.

In order to automate this process, we used TCP sockets to communicate with the server.

Password cracker script

```
import socket
import string
import os

# Set server and port number
HOST = "10.21.235.179"
PORT = 5555

# Setup hash constants
PRIME = 19
SEARCH_LETTERS = string.ascii_lowercase+"{}=_ "

# start with empty password
current_baseline_password = "-"*PRIME

# find the first PRIME letters of the password
# rest of the letters do not matter
for i in range(PRIME):
    for letter in SEARCH_LETTERS:
        # try each character in the search space for the current position
        current_try_password = list(current_baseline_password)
        current_try_password[(7*i+4)%PRIME] = letter
        sent_password = ''.join(current_try_password)
        print("Current try: {}".format(sent_password))
        sent_password += os.linesep
    # Server closes the TCP connection after a single try (Broken Pipe error)
    # Need to create a new socket for each guess.
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
        sock.connect((HOST, PORT))
        # Set a large timeout to account for large delays for correct
        # password
        sock.settimeout(100)
        prompt = sock.recv(1024)
        # Check if response received from server is as expected
        if prompt.strip() == b"Enter the password:":
            sock.send(bytes(sent_password, "UTF-8"))
            response = sock.recv(1024)
            # Python hackism to extract the float number
            time_taken = float(str(response).split("=")[1].split("\n")[0])
            # Incorrect guess for ith position - i+1 sec
            # Correct guess for ith position - i+2 sec
            if time_taken > i+2:
                current_baseline_password = sent_password.strip()
                break
    sock.close()
```

2.2 Results

Parameter	Value
Password	spm{fastandfurious}
Guesses	153
Colliding passwords	spm{fastandfurious}_hacked
	spm{fastandfurious}_h4ckeD
	spm{fastandfurious}c011iDe
	spm{fastandfurious}_collide
	spm{fastandfurious}_fA5t9

Tab. 1: Summary

```

aklsh@zephyrus CS6630/A1 on 12 main via  pyenv 3.9.5 (python-3.9.5)
^ nc 10.21.235.179 5555
Enter the password:
spm{fastandfurious}
Access Granted
Time taken to verify = 19.019651034963317
^C

aklsh@zephyrus CS6630/A1 on 12 main via  pyenv 3.9.5 (python-3.9.5) took 33s
^

```

Fig. 2: Password cracked and access granted