تمرین اول امنیت شبکه

محمد مهدی صمدی - 810101465

سوال اول:

در روش OTP کلیدی به طول plain text داریم. هر کاراکتر (8 بیت) با کاراکتر متناظرش در کلید جمع شده و پیمانه 26 گرفته می‌شود. در این سوال چون با یک کلید دو رمزنگاری انجام شده دیگر unconditional secure نیست.

$$C_1 \equiv P_1 + K,\ C_2 \equiv P_2 + K \quad \rightarrow \quad C_2 - C_1 \equiv (P_2 + K) - (P_1 - K) \equiv P_2 - P_1$$

طبق رابطه بالا برای کاراکتر iام هر کدام از ورودی و خروجی ها باید اختلاف کاراکتر در plain text ها برابر همان اختلاف در cipher text ها باشد.

```
c1 = 11111001011110011100110000010111 10000110

c2 = 11111010011001111101110100001001 10001000
```

- دو cipher text در چپ‌ترین کاراکتر یکی اختلاف دارند:

$$C_2 - C_1 \equiv 2'b11111010 - 2'b11111001 \equiv 2'b00000001 \equiv 1$$

پس باید در plain text ها نیز این رابطه برقرار باشد. حالت الف مشکلی ندارد ($b - a = 1$) اما حالت ب این شرط را نقض می‌کند ($g - d = 3$).

- دو cipher text در راست‌ترین کاراکتر دو عدد اختلاف دارند:

$$C_2 - C_1 \equiv 2'b10001000 - 2'b10000110 \equiv 2'b00000010 \equiv 2$$

پس باید در plain text ها نیز این رابطه برقرار باشد. حالت ب که در بررسی قبلی رد شده بود. اما حالت الف هم این شرط را برقرار نمی‌کند ($o - a = 14$).

پس هیچ‌کدام از دو حالت الف و ب درست نمی‌باشند.

سوال دوم:

در روش Base64 هر 6 بیت را یک به یک به کاراکتری طبق این جدول مپ می‌کنیم. جدول $2^6 = 64$ خانه دارد که با 52 حرف انگلیسی (کوچک و بزرگ)، 10 رقم 0-9 و دو کاراکتر + و / پر می‌شود. حال رشته hellojello را به حالت باینری در آورده و هر 6 بیت را جدا کرده و کاراکتر متناظرش از جدول را پیدا می‌کنیم.

$$011010, 000110, 010101, 101100, 011011, 000110, 111101, 101010, 011001, 010110, 110001, 101100, 011011, 110000$$

$$a \ , \ G \ , \ V \ , \ s \ , \ b \ , \ G \ , \ 9 \ , \ q \ , \ Z \ , \ W \ , \ x \ , \ s \ , \ b \ , \ w$$

متن کد شده به صورت $aGVsbG9qZWxsbW$ شد. این الگوریتم در نهایت هر 4 حرف $(4 \times 6 = 24bit)$ را با هم ذخیره می‌کند. در صورتی که تعداد کاراکترها بر 4 بخش‌پذیر نباشد، باید = به انتهای آن اضافه کنیم. پس در نهایت خواهیم داشت:

$$aGVs, bG9q, ZWxs, bW \ ==$$

سوالات سوم و چهارم:

کدهای این سوال و سوال بعد در فایل `Q3_4_solution.py` قرار دارند. مختصرا عملکرد آنان را توضیح می‌دهم:

کلاس DES الگوریتم را پیاده‌سازی می‌کند. در ابتدا یک بیت وکتور کلید و یک متغیر بولین برای اینکه مشخص کند از s-box های رندم یا استاندارد استفاده شود، به عنوان ورودی می‌گیرد.

متود `init_permutation_lists` لیست‌های permutation را مقداردهی می‌کند.

متود `initially_process_key` برای تبدیل کلید 64 بیتی به کلید 56 بیتی و permutation روی آن است.

متود `generate_round_keys` کلیدهای هر راند را ساخته و در لیستی ذخیره می‌کند.

متود `generate_sboxes` با توجه به بولین ورودی یا آن‌ها را random پر می‌کند یا با مقادیر دیفالت داده شده.

```
def __init__(self, key: BitVector, use_standard_sboxes: bool = True):
    self.BLOCK_SIZE = 64
    self.NUMOF_DES_ROUNDS = 16
    self.NUMOF_SBOXES = 8
    self.key = key
    self.init_permutation_lists()
    self.initially_process_key()
    self.generate_round_keys()
    self.generate_sboxes(use_standard_sboxes=use_standard_sboxes)
```

متود زیر یک راند از الگوریتم را اجرا می‌کند. ابتدا ورودی دو بخش می‌شود. بخش سمت راست از 32 بیت به 48 بیت گسترش داده می‌شود. حال با کلید راند xor شده و سپس از s-box ها و p-box می‌گذرد. در نهایت با بخش چپ xor می‌شود. تا اینجا بخش راست خروجی را ساخته‌ایم. بخش چپ آن همان بخش راست اولیه خواهد بود.

```
def simulate_DES_round(self, LE: BitVector, RE: BitVector, round_key:
BitVector):
    newLE = RE
    expandedRE = RE.permute(permute_list=self.EXPANSION_PERMUTATION)
    xor_out = expandedRE ^ round_key
    substitution_out = self.substitute(xor_out)
    permutation_out = substitution_out.permute(self.PBOX_PERMUTATION)
    newRE = permutation_out ^ LE
    return (newLE, newRE)
```

متود زیر تمام راند ها را به اضافه permutation اولیه و انتهایی انجام می‌دهد.

```
def simulate_DES(self, bv: BitVector):
    bv = bv.permute(permute_list=self.INITIAL_PERMUTATION)
    (LE, RE) = bv.divide_into_two()
    for round_key in self.round_keys:
        (LE, RE) = self.simulate_DES_round(LE=LE, RE=RE, round_key=round_key)
    bv = RE + LE
    bv = bv.permute(permute_list=self.FINAL_PERMUTATION)
    return bv
```

متود زیر عمل encryption را روی یک ورودی انجام می‌دهد. برای عمل decryption تنها باید ترتیب کلید راند ها را برعکس کنیم (با متود `switch_mode`) و سپس از همین متود استفاده کنیم.

```
def encrypt(self, file: BitVector):
    result = None
    while file.more_to_read:
        bv = file.read_bits_from_file(self.BLOCK_SIZE)

        if bv.length() < self.BLOCK_SIZE:
            bv.pad_from_right(self.BLOCK_SIZE - bv.length())

        if result is None:
            result = self.simulate_DES(bv=bv)
        else:
            result = result + self.simulate_DES(bv=bv)

    return result
```

تا به اینجا الگوریتم پیاده‌سازی شده است. برای اجرای آن کدهایی نوشته شده‌اند که آن‌ها را هم بررسی می‌کنیم.

متود `receive_encryption_key` برای دریافت ورودی کلید از کاربر و چک کردن درستی اندازه آن است.

متود `clear_files` برای پاک کردن فایل‌های خروجی هر بار اجرای الگوریتم می‌باشد.

متود زیر برای محاسبه اثر avalanche روی دو بیت وکتور می‌باشد. ابتدا دو وکتور را xor می‌کند تا بیت‌[ایی که اختلاف دارند مشخص شود. سپس با `BitVector.count_bits` می‌توان تعداد بیت‌های 1 را شمرد.

```
def avalanche_effect(bv1: BitVector, bv2: BitVector, print_result=False):
    diff = bv1 ^ bv2
    changed_bits = diff.count_bits()
    total_bits = bv1.length()
    return (changed_bits / total_bits) * 100
```

در نهایت سه متود برای تست برنامه نوشته شده‌اند.

متود `run_encryption_decryption` ابتدا یک ورودی را code و سپس حاصل را decode می‌کند تا از صحت الگوریتم مطمئن شویم.

متود `run_with_different_sboxes` یکبار با حالت s-box های استاندارد و بار دیگر با حالت random اجرا می‌کند.

متود `run_with_different_inputs` با دو ورودی مختلف تست می‌کند. ورودی دوم همان ورودی اول می‌باشد که در هر block آن دقیقا یک بیت عوض شده است.

متود دوم و سوم برای بررسی اثر avalanche استفاده می‌شوند. هر کدام به تعداد مشخص (در اجرای زیر 1000) دفعه اجرا شده و میانگین درصدشان گزارش می‌شود که طبق نتیجه زیر به عدد 50 درصد مورد انتظار رسیده‌ایم.

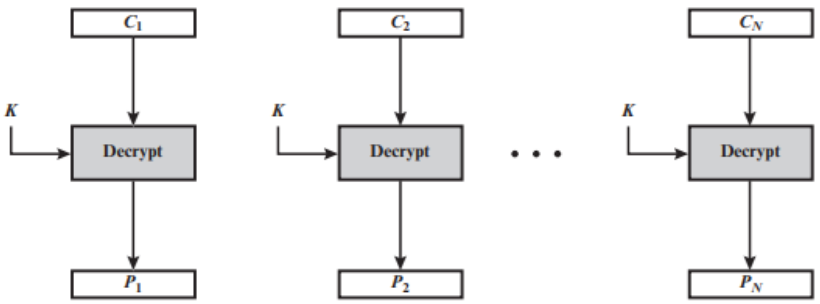
```
PS F:\UT\Network Security\HWs\HW1\HW1_starter_code> py .\Q3_4_solution.py
Running 1000 tests with different s-boxes...
Please enter an encryption key consisting of 8 printable characters: abcdefgh
Average Avalanche effect: 49.95

Running 1000 tests with different inputs...
Please enter an encryption key consisting of 8 printable characters: abcdefgh
Average Avalanche effect: 50.05
```

سوال پنجم:

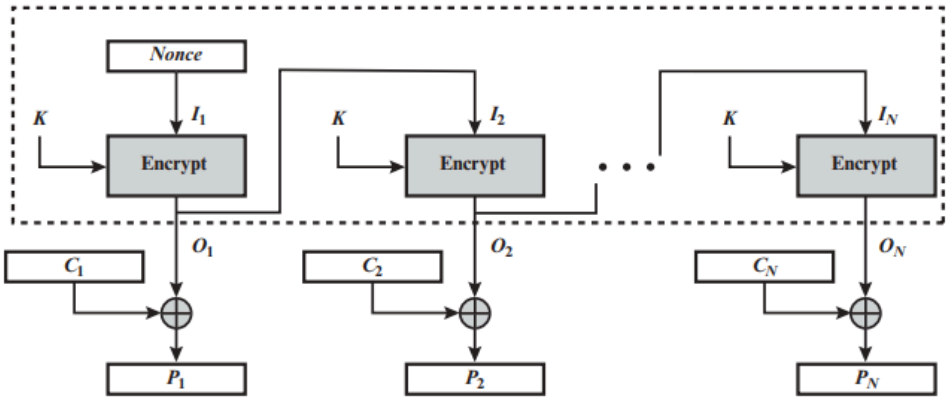
ابتدا نحوه عملکرد هر کدام از چهار الگوریتم را بررسی می‌کنیم.

ECB:



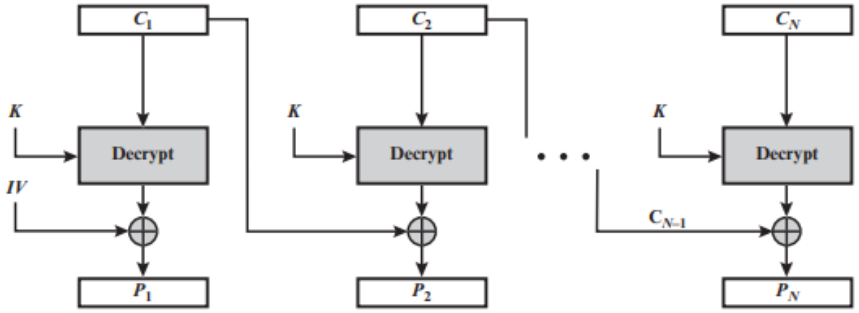
واضح است که هر block مستقل از بقیه encrypt/decrypt می‌شود. پس ارور در یک block منجر به اشتباه در همان block می‌شود و به بقیه سرایت نمی‌کند.

OFB:



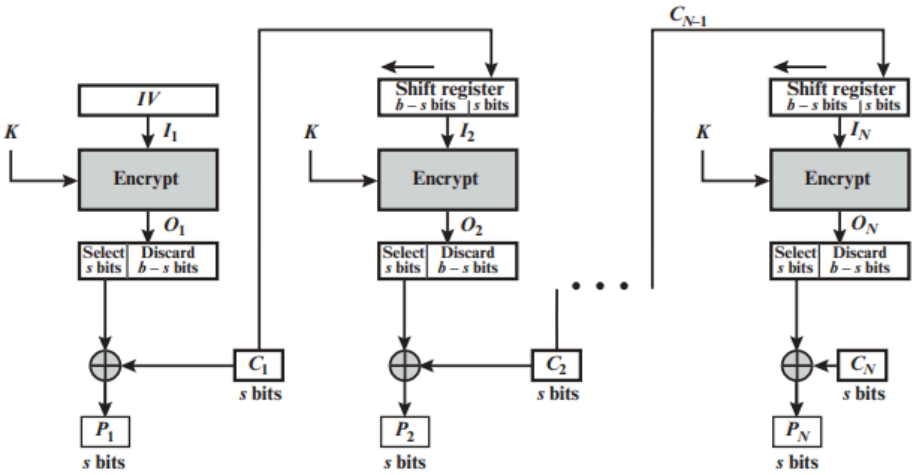
در این مدل کلیدها به صورت sequential ترکیب می‌شوند اما خود text ها مستقل از هم می‌باشند. پس دوباره ارور در یک block به بقیه سرایت نمی‌کند.

CBC:



طبق تصویر ارور در C_i تنها خروجی P_i و P_{i+1} را تحت تاثیر می‌گذارد.

CFB:



طبق تصویر همانند الگوریتم قبل، ارور در C_i تنها خروجی P_i و P_{i+1} را تحت تاثیر می‌گذارد.