CS152-Homework4

Hongchen Cao 2019533114

2020.10.6

1

1° Since $PC1$, $PC2$ and operation $<$ and $<<$ are determininistinc, bitwise permutataions, where the value of the input bits don't affect where they're mapped to, $PC1(c(k)) = c(PC1(k))$. As such, the subkeys which are given into f for $c(k)$ are guaranteed to be the bitwise complement of those which would have been given into f for k in all cases.

2° Then, IP is a bit-order permutation, so $IP(c(x)) = c(IP(x))$ is true.

3° Then, prove $E(c(x)) = c(E(x))$. What we need to consider is whether it matters if the bits are flipped before or after being mapped to the two locations in the expanded vector. There are no combinatory operations here, a single bit in the expanded vector always has one original bit in original vector. So, flipping one bit prior to mapping, or flipping the two copies after the mapping, makes no difference to the outcome.

4° Then, prove if R^{i-1}, L^{i-1}, k^i generate R^i, then $c(R^{i-1}), c(L^{i-1}), c(k^i)$ generate $c(R^i)$. $P(S(E(c(R_0))\oplus c(k_1))) = P(S(c(E(R_0)) \oplus c(k_1))) = P(S(E(R_0) \oplus k_1))$. So $f(R_0, k_1) = f(c(R_0), c(k_1))$. $c(L_0) \oplus f(R_0, k_1) = c(L_0 \oplus f(R_0, k_1)) = c(L_0) \oplus f(R_0, k_1)$. Now that we've proved that inverting the key and input will produce bitwise inverted output after one round, so by induction we know this is same for any round after.

5° Finally, we need to prove $IP^{-1}(c(x)) = c(IP^{-1}(x))$. Since this is the inverse of IP and is also simple bitwise permutations, it is true.

Thus, we know that $y' = c(y)$.

2

2.1

```
sBox = [
    [0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7,
        0xAB, 0x76],
    [0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4,
        0x72, 0xC0],
    [0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8,
        0x31, 0x15],
    [0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27,
        0xB2, 0x75],
    [0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3,
        0x2F, 0x84],
    [0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C,
        0x58, 0xCF],
    [0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C,
        0x9F, 0xA8],
    [0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF,
        0xF3, 0xD2],
    [0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D,
        0x19, 0x73],
    [0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E,
        0x0B, 0xDB],
    [0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95,
        0xE4, 0x79],
    [0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A,
        0xAE, 0x08],
    [0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD,
        0x8B, 0x8A],
    [0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1,
        0x1D, 0x9E],
    [0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55,
        0x28, 0xDF],
    [0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54,
        0xBB, 0x16]
]
rCon = [0x1000000, 0x2000000, 0x4000000, 0x8000000, 0x10000000, 0x20000000, 0x40000000,
        0x80000000, 0x1b000000,
        0x36000000]
mixColBox = [[2, 3, 1, 1], [1, 2, 3, 1], [1, 1, 2, 3], [3, 1, 1, 2]]


def hexXor(a: str, b: str) -> str:
    result = int(a, 16) ^ int(b, 16)  # convert to integers and xor them together
    return '{:08x}'.format(result)  # convert back to hexadecimal


def subWord(subWord: str) -> str:
    res = ''
```

```python
    for i in range(4):
        temp = sBox[int(subWord[i * 2], 16)][int(subWord[i * 2 + 1], 16)]
        temp = '{:02x}'.format(temp)
        res += temp
    return res


def rotWord(subWord: str) -> str:
    return subWord[2:] + subWord[0:2]


def keyExpansion(key: str) -> list:
    word = list()
    res = list()
    # initialize w0-w3
    for i in range(4):
        word.append(key[8 * i:8 * i + 8])
    # generate w4-w43
    for i in range(4, 44):
        temp = (word[i - 1])
        if i % 4 == 0:
            temp = hexXor(subWord(rotWord(temp)), '{:x}'.format(rCon[i // 4 - 1]))
        word.append(hexXor(word[i - 4], temp).upper())
    # generate keys
    for i in range(11):
        temp = list()
        for j in range(4):
            temp.append(word[i * 4 + j][0:2] + " " + word[i * 4 + j][2:4] + " " + \
                        word[i * 4 + j][4:6] + " " + word[i * 4 + j][6:8])
        res.append(temp)

    return res
```

Output(key):

Key0:

w_0: 2B7E1516 w_1: 28AED2A6 w_2: ABF71588 w_3: 09CF4F3C

Key1:

w_4: A0FAFE17 w_5: 88542CB1 w_6: 23A33939 w_7: 2A6C7605

Key2:

w_8: F2C295F2 w_9: 7A96B943 w_{10}: 5935807A w_{11}: 7359F67F

Key3:

w_{12}: 3D80477D w_{13}: 4716FE3E w_{14}: 1E237E44 w_{15}: 6D7A883B

Key4:

w_{16}: EF44A541 w_{17}: A8525B7F w_{18}: B671253B w_{19}: DB0BAD00

Key5:

w_{20}: D4D1C6F8 w_{21}: 7C839D87 w_{22}: CAF2B8BC w_{23}: 11F915BC

Key6:

w_{24}: 6D88A37A w_{25}: 110B3EFD w_{26}: DBF98641 w_{27}: CA0093FD

Key7:

w_{28}: 4E54F70E w_{29}: 5F5FC9F3 w_{30}: 84A64FB2 w_{31}: 4EA6DC4F

Key8:

w_{32}: EAD27321 w_{33}: B58DBAD2 w_{34}: 312BF560 w_{35}: 7F8D292F

Key9:

w_{36}: AC7766F3 w_{37}: 19FADC21 w_{38}: 28D12941 w_{39}: 575C006E

Key10:

w_{40}: D014F9A8 w_{41}: C9EE2589 w_{42}: E13F0CC8 w_{43}: B6630CA6

2.2

```python
def subBytes(text: str) -> str:
    res = str()
    for i in range(4):
        res += subWord(text[8 * i:8 * i + 8])
    return res


def shiftRows(text: str) -> str:
    return text[0:2] + text[10:12] + text[20:22] + text[30:32] + text[8:10] + text[18:20]
        + text[28:30] + text[6:8] + \
           text[16:18] + text[26:28] + text[4:6] + text[14:16] + text[24:26] + text[2:4]
        + text[12:14] + text[22:24]


def galoisMult(a, b):
    # Multiplication in the Galois field GF(2^8).
    p = 0
    hi_bit_set = 0
    for i in range(8):
        if b & 1 == 1:
            p ^= a
        hi_bit_set = a & 0x80
        a <<= 1
        if hi_bit_set == 0x80:
            a ^= 0x1b
        b >>= 1
    return p % 256


def mixColumns(text: str) -> str:
    res = str()
    for i in range(4):
        for j in range(4):
            a = '{:02x}'.format(galoisMult(mixColBox[j][0], int(text[8 * i:8 * i + 2], 16
        )))
            b = '{:02x}'.format(galoisMult(mixColBox[j][1], int(text[8 * i + 2:8 * i + 4]
        , 16)))
```

```
            c = '{:02x}'.format(galoisMult(mixColBox[j][2], int(text[8 * i + 4:8 * i + 6]
                                                                    , 16)))
            d = '{:02x}'.format(galoisMult(mixColBox[j][3], int(text[8 * i + 6:8 * i + 8]
                                                                    , 16)))
            e = hexXor(a, b)
            f = hexXor(c, d)
            res += hexXor(e, f)
    return res


def encryption(plaintext: str, key: str) -> str:
    # initialization
    ciphertext = str()
    keyList = keyExpansion(key)
    keyStr = list()
    for li in keyList:
        temp = str()
        for word in li:
            temp += word.replace(" ", '')
        keyStr.append(temp)
    for i in range(4):
        ciphertext += hexXor(plaintext[i * 8:i * 8 + 8], keyStr[0][i * 8:i * 8 + 8])
    # nine rounds
    for i in range(10):
        ciphertext = subBytes(ciphertext)
        ciphertext = shiftRows(ciphertext)
        if not i == 9:
            ciphertext = mixColumns(ciphertext)
        # AddRoundKey
        temp = str()
        for j in range(4):
            temp += hexXor(ciphertext[j * 8:j * 8 + 8], keyStr[i + 1][j * 8:j * 8 + 8])
        ciphertext = temp
    print(ciphertext)
```

Output(ciphertext):

39 25 84 1D 02 DC 09 FB DC 11 85 97 19 6A 0B 32

3

3.1

Output(Linear approximation table):

a	b															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	16	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
1	8	10	6	8	10	8	8	6	4	6	6	8	10	8	4	10
2	8	10	8	10	6	8	6	8	6	8	10	4	4	6	8	10
3	8	8	10	10	8	12	10	6	6	6	8	8	10	6	12	8
4	8	10	8	6	8	10	8	6	10	4	10	8	6	8	6	4
5	8	12	6	6	10	10	8	12	6	10	8	8	8	8	10	6
6	8	8	12	8	10	10	6	10	8	8	8	12	6	6	6	10
7	8	6	6	8	12	6	10	8	8	6	6	8	4	6	10	8
8	8	10	10	8	8	6	6	8	10	8	4	6	10	4	8	6
9	8	8	8	12	10	10	6	10	10	6	6	6	8	12	8	8
10	8	12	10	10	6	6	12	8	8	8	6	10	6	10	8	8
A	8	6	12	6	8	6	8	10	4	6	8	6	8	10	8	6
B	8	8	10	10	12	8	10	6	8	12	10	6	8	8	6	6
C	8	6	8	6	6	12	10	8	8	10	4	6	6	8	6	8
D	8	6	6	12	6	8	8	10	6	8	8	10	8	6	6	4
E	8	8	8	8	8	8	12	12	10	6	10	6	10	6	6	10

```python
sBox = [0x8, 0x4, 0x2, 0x1, 0xC, 0x6, 0x3, 0xD, 0xA, 0x5, 0xE, 0x7, 0xF, 0xB, 0x9, 0x0]
SBox = ['{:04b}'.format(num) for num in sBox]


def binMul(a: str, b: str) -> str:
    res = 0
    for i in range(len(a)):
        temp = int(a[i]) * int(b[i])
        res = (res + temp) % 2
    return res


def calculateNL(a: str, b: str) -> int:
    res = 0
    for i in range(16):
        y = SBox[i]
        x = '{:04b}'.format(i)
        if (binMul(a, x) + binMul(b, y)) % 2 == 0:
            res += 1
    return res


def generateTable():
    for i in range(16):
        for j in range(16):
            print(calculateNL('{:04b}'.format(i), '{:04b}'.format(j)), end='\t')
```

3.2

$S_4^1 : T_1 = U_{16}^1 \oplus V_{13}^1 = -\frac{1}{4}$

$S_1^2 : T_2 = U_4^2 \oplus V_1^2 = -\frac{1}{4}$

$S_1^3 : T_3 = U_1^3 \oplus V_1^3 \oplus V_3^3 = -\frac{1}{4}$

$T_1 \oplus T_2 \oplus T_3 = -\frac{1}{16}$

From

$U_{16}^1 = X_{16} \oplus K_{16}^1$

$U_4^2 = V_{13}^1 \oplus K_4^2$

$U_1^3 = V_1^2 \oplus K_1^3$

$U_1^4 = V_1^3 \oplus K_1^4$

$U_9^4 = V_3^3 \oplus K_9^4$

we have $X_{16} \oplus U_1^4 \oplus U_9^4 = T_1 \oplus T_2 \oplus T_3 = \pm\frac{1}{16}$

3.3

Algorithm 1: LINEARATTACK

Input: $\mathcal{T}$, T, $\pi_{S'}^{-1}$

Output: $maxkey$

for $(L_1, L_2) \leftarrow (0, 0)$ *to* (F, F) **do**
    $Count[L_1, L_2] \leftarrow 0$
end
foreach $(x, y) \in \mathcal{T}$ **do**
    for $(L_1, L_2) \leftarrow (0, 0)$ *to* (F, F) **do**
        $v_{(1)}^4 \leftarrow L_1 \oplus y_{(1)}$
        $v_{(3)}^4 \leftarrow L_2 \oplus y_{(3)}$
        $u_{(1)}^4 \leftarrow \pi_{S'}^{-1}(v_{(1)}^4)$
        $u_{(3)}^4 \leftarrow \pi_{S'}^{-1}(v_{(3)}^4)$
        $z \leftarrow x_{16} \oplus u_1^4 \oplus u_9^4$
        if $z = 0$ **then**
            $Count[L_1, L_2] \leftarrow Count[L_1, L_2] + 1$
        end
    end
end
$max \leftarrow -1$
for $(L_1, L_2) \leftarrow (0, 0)$ *to* (F, F) **do**
    $Count[L_1, L_2] \leftarrow |Count[L_1, L_2] - T/2|$
    if $Count[L_1, L_2] > max$ **then**
        $max \leftarrow Count[L_1, L_2]$
        $maxkey \leftarrow (L_1, L_2)$
    end
end

3.4

```
S = {0: 0x8, 1: 0x4, 2: 0x2, 3: 0x1, 4: 0xC, 5: 0x6, 6: 0x3, 7: 0xD, 8: 0xA, 9: 0x5, 0xA:
                                                         0xE, 0xB: 0x7, 0xC: 0xF,
     0xD: 0xB, 0xE: 0x9, 0xF: 0x0}


def hexXor(a: str, b: str) -> str:
    result = int(a, 16) ^ int(b, 16)  # convert to integers and xor them together
    return '{:08x}'.format(result)  # convert back to hexadecimal


def linearAttack(P, T, Sb):
    count = {}
    for i in Sb.keys():
        for j in Sb.keys():
            count[i + j] = 0
    # get pi_{s'}^{-1}
    sReverse = {}
    for item in Sb.items():
        sReverse[item[1]] = item[0]

    for i in range(T):
        x = P[0][i]
        y = P[1][i]
        for L in count.keys():
            v41 = hexXor(L[0], y[0])
            v43 = hexXor(L[1], y[2])
            u41 = sReverse[v41]
            u43 = sReverse[v43]
            z = (binary[x[3][3]] + binary[u41][0] + binary[u43][0]) % 2
            if z == 0:
                count[L] += 1

    maxValue = -1

    for L in count.keys():
        count[L] = abs(count[L] - T / 2)
        if count[L] > maxValue:
            maxValue = count[L]
            maxkey = (L[0], L[1])
    print(maxkey)
```

Output:

E,C

4

4.1

a	b															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	2	0	4	0	2	0	0	0	0	2	4	0	0	2
2	0	2	0	0	0	0	0	2	0	0	2	0	0	2	2	6
3	0	2	0	4	0	2	0	0	0	2	0	4	0	2	0	0
4	0	4	2	2	0	2	0	2	2	0	0	2	0	0	0	0
5	0	0	0	4	0	0	0	4	0	0	0	0	2	2	2	2
6	0	0	0	0	2	0	2	0	2	0	2	0	2	2	2	2
7	0	0	4	2	2	0	0	0	4	2	0	0	0	0	2	0
8	0	2	0	0	2	4	2	2	0	2	0	0	0	2	0	0
9	0	6	0	0	0	0	2	0	0	0	2	4	0	2	0	0
A	0	0	2	0	0	0	0	2	4	0	4	2	0	0	2	0
B	0	0	0	0	2	2	2	2	0	0	0	0	4	0	4	0
C	0	0	2	0	0	2	4	0	2	0	0	0	2	2	2	0
D	0	0	2	2	2	0	2	0	0	2	6	0	0	0	0	0
E	0	0	2	2	0	0	0	0	2	6	0	0	0	0	0	4
F	0	0	0	0	2	4	0	2	0	2	0	2	2	2	0	0

Table 1: Dr.Hibbert

4.2

$S_1^1 : R_p(1001, 0001) = \frac{3}{8}$
$S_4^1 : R_p(1001, 0001) = \frac{3}{8}$
$S_4^2 : R_p(1001, 0001) = \frac{3}{8}$
$S_4^3 : R_p(0001, 1100) = \frac{1}{4}$
$R_p(1001\,0000\,0000\,1001,\ 0000\,0000\,0000\,1100) = \frac{27}{2048}$
Thus,
$x' = 1001\,0000\,0000\,1001 \rightarrowtail (u^4)' = 0001\,0001\,0000\,0000$

4.3

Algorithm 2: DIFFERENTIALATTACK

Input: $\mathcal{T}$, T, $\pi_{S''}^{-1}$

Output: $maxkey$

for $(L_1, L_2) \leftarrow (0,0)$ *to* (F,F) **do**
    $Count[L_1, L_2] \leftarrow 0$
end
foreach $(x, y, x^*, y^*) \in \mathcal{T}$ **do**
    if $(y_{(3)} = (y_{(3)})^*)$ *and* $(y_{(4)} = (y_4)^*)$ **then**
        for $(L_1, L_2) \leftarrow (0,0)$ *to* (F,F) **do**
            $v^4_{(1)} \leftarrow L_1 \oplus y_{(1)}$
            $v^4_{(2)} \leftarrow L_1 \oplus y_{(2)}$
            $u^4_{(1)} \leftarrow \pi_{S'}^{-1}(v^4_{(1)})$
            $u^4_{(2)} \leftarrow \pi_{S'}^{-1}(v^4_{(2)})$
            $(v^4_{(1)})^* \leftarrow L_1 \oplus (y_{(1)})^*$
            $(v^4_{(2)})^* \leftarrow L_1 \oplus (y_{(2)})^*$
            $(u^4_{(1)})^* \leftarrow \pi_{S'}^{-1}((v^4_{(1)})^*)$
            $(u^4_{(2)})^* \leftarrow \pi_{S'}^{-1}((v^4_{(2)})^*)$
            $(u^4_1)' \leftarrow u^4_{(1)} \oplus (u^4_{(1)})^*$
            $(u^4_2)' \leftarrow u^4_{(2)} \oplus (u^4_{(2)})^*$
            if $(u^4_1)' = 0001$ *and* $(u^4_2)' = 0001$ **then**
                $Count[L_1, L_2] \leftarrow Count[L_1, L_2] + 1$
            end
        end
    end
end
$max \leftarrow -1$
for $(L_1, L_2) \leftarrow (0,0)$ *to* (F,F) **do**
    if $Count[L_1, L_2] > max$ **then**
        $max \leftarrow Count[L_1, L_2]$
        $maxkey \leftarrow (L_1, L_2)$
    end
end

4.4

```
sBox = {0: 0xE, 1: 0x2, 2: 0x1, 3: 0x3, 4: 0xD, 5: 0x9, 6: 0x0, 7: 0x6, 8: 0xF, 9: 0x4,
                                                0xA: 0x5, 0xB: 0xA, 0xC: 0x8,
        0xD: 0xC, 0xE: 0x7, 0xF: 0xB}
```

```python
def hexXor(a: str, b: str) -> str:
    result = int(a, 16) ^ int(b, 16)  # convert to integers and xor them together
    return '{:08x}'.format(result)  # convert back to hexadecimal


def diffAttack(P, T, Sb):
    count = {}
    for i in Sb.keys():
        for j in Sb.keys():
            count[i + j] = 0

    for i in range(T):
        x = P[0][i]
        y = P[1][i]
        xs = P[2][i]
        ys = P[3][i]
        if y[2] == ys[2] and y[3] == ys[3]:
            for L in count.keys:
                v41 = hexXor(L[0], y[0])
                v42 = hexXor(L[1], y[1])
                u41 = Sb[v41]
                u42 = Sb[v42]
                v41s = hexXor(L[0], ys[0])
                v42s = hexXor(L[1], ys[1])
                u41s = Sb[v41s]
                u42s = Sb[v42s]
                u41pi = hexXor(u41, u41s)
                u42pi = hexXor(u42, u42s)
                if binary[u41pi] == 0b0001 and binary[u42pi] == 0b0001:
                    count[L] += 1

    maxValue = -1

    for L in count.keys():
        if count[L] > maxValue:
            maxValue = count[L]
            maxkey = (L[0], L[1])
    print(maxkey)
```

Output:

E, C