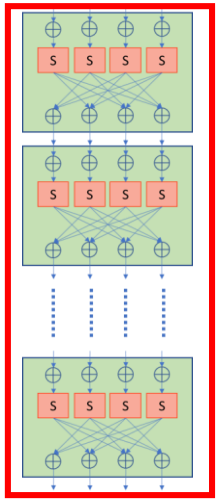
Cryptanalysis
(암호분석)

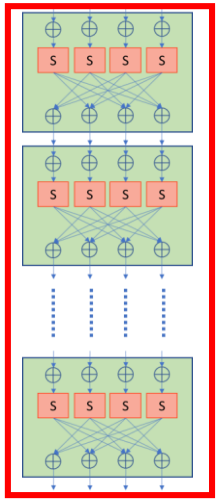
Exhaustive Key Search
Meet In The Middle(MITM) Attack

2023.4

Contents

- Integer ↔ List
- Pickle dump (variable save/load)
- Brute force attack: Exhaustive key search
- Meet-in-the-Middle Attack

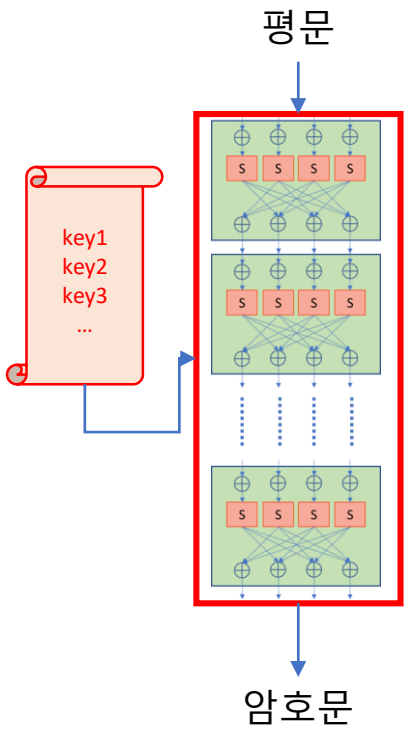




Brute Force Attack (1)

- 암호키 전수조사 (Exhaustive key search)
  - 모든 키를 시도해보는 방법으로 공격
  - 암호키의 크기가 작은 경우만 가능

- 예: TC20의 공격 (기지평문 공격)
  - TC20: 블록크기=키크기=32비트
  - 주어진 평문, 암호문으로부터 암호키를 찾는 공격
  - 공격 알고리즘
    - 2^{32}가지 암호키 모두를 대입하여 평문을 암호화
    - 주어진 암호문과 같은 것이 나오면 올바른 암호키 후보로 선택
    - False alarm 확률: $1/2^{32}$ (잘못된 키가 암호키 후보가 될 확률)



Int2list() – 정수를 리스트로

0x12345678: 16진수표현

```
print(0x12345678)   → 305419896
print(hex(305419896)) → 0x12345678
```

```
#--- int(4bytes) to list
#--- Example: 0x12345678 -> [ 0x12, 0x34, 0x56, 0x78 ]

def int2list(n):
    out_list = []
    out_list.append( (n >> 24) & 0xff )
    out_list.append( (n >> 16) & 0xff )
    out_list.append( (n >>  8) & 0xff )
    out_list.append( (n      ) & 0xff )

    return out_list
```

비트 마스크 (선택 비트만 추출)

```
0x00123456 & 0xff = 56
0x00001234 & 0xff = 34
0x00000012 & 0xff = 12

0xff = 0x000000ff
& : 비트연산자 AND
```

비트 쉬프트 (오른쪽으로 밀기)

```
0x12345678 >>  8 = 0x00123456
0x12345678 >> 16 = 0x00001234
0x12345678 >> 24 = 0x00000012
```

list2int() – 리스트를 정수로

```
#--- list to int
#--- Example: [ 0x12, 0x34, 0x56, 0x78 ] -> 0x12345678

def list2int(l):
    n = 0
    num_bytes = len(l)
    for idx in range(len(l)):
        n += l[idx] << 8*(num_bytes - idx -1)

    return n
```

예제: 함수 입력 l = [0x12, 0x34, 0x56, 0x78]

```
num_bytes = len(l) = 4
```

정수로 만들기

```
n= 0
n = 0x00000000 + (0x12 << 24) = 0x12000000
n = 0x12000000 + (0x34 << 16) = 0x12340000
n = 0x12340000 + (0x56 <<  8) = 0x12345600
n = 0x12345600 + (0x78 <<  0) = 0x12345678
```

비트 쉬프트 (왼쪽으로 밀기)

```
Shift = 8*(num_bytes - idx -1)

idx = 0  → Shift = 8*(4 - 0 - 1) = 24
idx = 1  → Shift = 8*(4 - 1 - 1) = 16
idx = 2  → Shift = 8*(4 - 2 - 1) =  8
idx = 3  → Shift = 8*(4 - 3 - 1) =  0
```

평문-암호문 만들기

- 키 전수조사 공격을 시험하기 위한 예제 만들기
  - 평문(pt), 암호키(key)를 설정하고 암호문(ct)을 만든다.
  - 키 전수조사 공격 프로그램에는 평문(pt), 암호문(ct)만을 입력으로 하여, 암호키를 찾아내도록 한다.

```
import TC20_Enc_lib

given_pt = [0, 1, 2, 3]
key = [0, 20, 20, 4]

ct = TC20_lib.TC20_Enc(given_pt, key)

print('pt  =', given_pt)
print('key =', key)
print('ct  =', ct)
```

암호는 Toy Cipher TC20을 사용한다.

(출력)

```
pt  = [0, 1, 2, 3]
key = [0, 20, 20, 4]
ct  = [192, 126, 66, 83]
```

Exhaustive Key Search

```
#key = [0, 20, 20, 4]

given_pt = [0, 1, 2, 3]
given_ct = [192, 126, 66, 83]

Flag = False
KeySize = 1<<24 # 24-bit key

print('Key Searching', end=' ')
for idx in range(0, KeySize):
    key_guess = int2list(idx)
    pt = TC20_lib.TC20_Dec(given_ct, key_guess)
    if pt == given_pt:
        Flag = True
        break
    if (idx % (1<<16)) == 0:
        print('.', end='')

print('\n')
if Flag:
    print('Key found!')
    print('Key =', key_guess)
else:
    print('Key Not found!')
```

Flag : for-loop 종료시 암호키를 찾았는지 확인하는 변수
KeySize : 키 전수조사 범위 설정용 (1<<24 = 2^{24})
(전체로 하면 너무 오래 걸림)

평문-암호문 쌍을 생성하는 암호키를 찾으면 for-loop를 나간다.

Exhaustive Key Search

- 생각해 볼 문제
  - ct = E(pt, key)를 만족하는 다른 키가 우연히 나올 확률은?
  - 키 후보가 나오면 모두 키 후보 리스트에 넣어 두는 방식을 구현하려면?
  - 여러 개의 키 후보가 발견되는 경우 올바른 키를 얻기 위해서는 어떤 방법을 추가로 사용할까?

Double Encryption

- 블록암호 안전성 문제
  - 암호키 전수조사가 가능하다면?

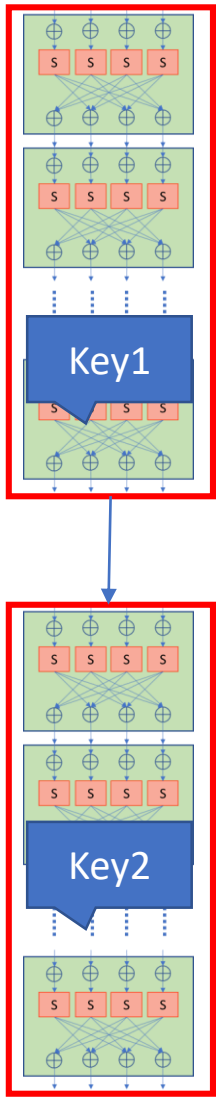
    → 암호키 크기를 늘이면 됨

DES의 56비트 암호키가 작다면,
2배로 늘이면 된다.

56비트 전수조사에 1초가 걸린다면,
112비트 전수조사에는23억년 걸린다.

- Double Encryption
  - 암호 알고리즘: C = E(P, key)
  - 강화된 알고리즘: C = E(E(P,key1), key2)
  - 블록 크기는 그대로, 암호키 크기는 2배로 강화

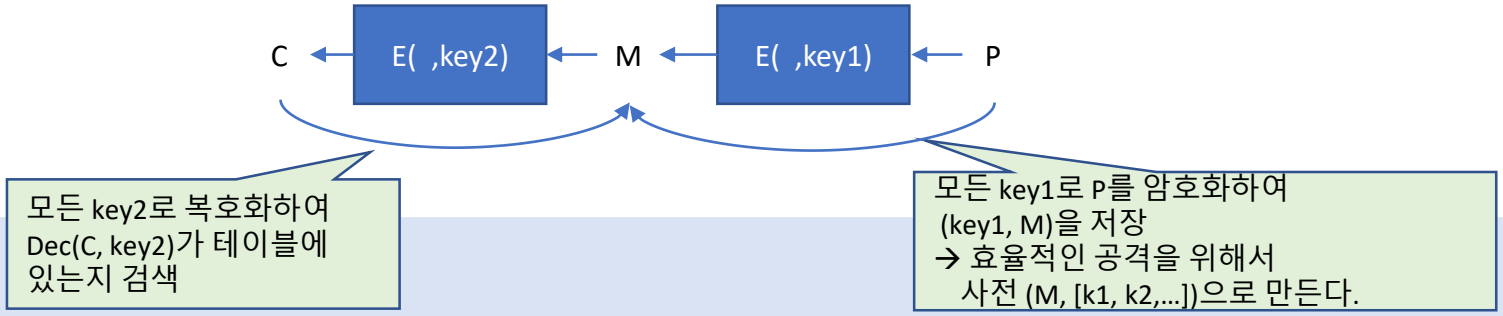
→ 키 전수조사 공격에 안전할까?



Brute Force Attack (2)

- Double Encryption의 안전성
  - C = E(P, key) 키 크기: 2^k → 공격량: 2^k 암호화 계산
  - C = E(E(P,key1), key2) 키 크기: 2^{2k} → 공격량: 2^{2k} 을 기대하지만
    → MITM attack을 이용하면 2^{k+1} 로 공격이 가능함
  - 결론적으로 double encryption의 안전성은 강화되지 않음

- MITM(Meet-in-the-Middle) Attack 알고리즘
  - 주어진 평문(P)와 암호문(C)에 대하여, 암호키 key1, key2를 찾는 공격
  - 평문(P)를 가능한 모든 key1으로 암호화하여 테이블에 저장한다.
  - 암호문 (C)를 가능한 모든 key2로 복호화하여 저장해둔 테이블에서 찾는다.



Double Encryption 구현

```
#- 암호키 (공격에서 찾아야 할 키)

key1 = [0, 20, 20, 4]
key2 = [0, 1, 2, 3]

#- 평문1-암호문1
pt1 = [0, 1, 2, 3]
mid1 = TC20_lib.TC20_Enc(pt1, key1)
ct1 = TC20_lib.TC20_Enc(mid1, key2)

print('pt1  =', pt1)
print('mid1 =', mid1)
print('ct1  =', ct1)

#- 평문2-암호문2
pt2 = [4,5,6,7]
mid2 = TC20_lib.TC20_Enc(pt2, key1)
ct2 = TC20_lib.TC20_Enc(mid2, key2)

print('pt2  =', pt2)
print('mid2 =', mid2)
print('ct2  =', ct2)
```

공격 프로그램에 찾아야 할 암호키: key1, key2

(예제 수행시간을 단축하기 위해 첫 바이트를 0으로 고정한다)

두 개의 (평문, 암호문) 쌍 (pt1, ct1), (pt2, ct2) 를 입력으로 암호키 key1, key2를 찾는 공격 예제를 만들어 본다.

(출력)

```
pt1  = [0, 1, 2, 3]
mid1 = [192, 126, 66, 83]
ct1  = [30, 18, 28, 114]

pt2  = [4, 5, 6, 7]
mid2 = [63, 233, 23, 246]
ct2  = [215, 173, 154, 71]
```

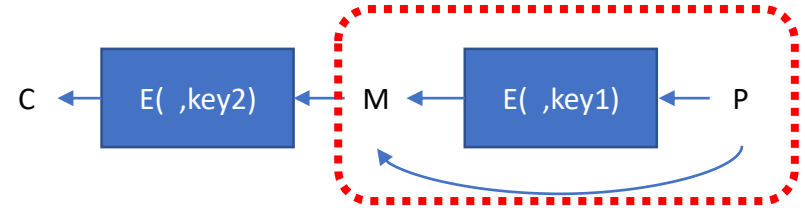
(키-암호문) 테이블 만들기

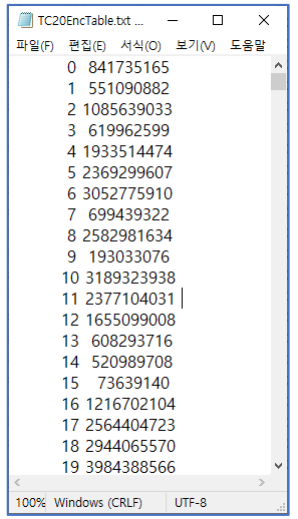
```
#- 주어진 평문을 모든 후보키로 암호한 결과를 파일로 저장한다.
def make_enc_table(pt):
    f = open('TC20EncTable.txt', 'w+')

    print('Encryption Table File', end=' ')
    # 전수조사 범위를 24비트 한정한다 key=[0,*, *, *]
    N = 1<<24

    for idx in range(0, N):
        key = int2list(idx)
        mid = TC20_lib.TC20_Enc(pt, key)
        int_key = list2int(key) # = idx
        int_mid = list2int(mid)

        str = format('%10d %10d \n' %(int_key, int_mid))
        f.write(str)

        if (idx % (1<<18)) == 0:
            print('.', end='')

    f.close()
```

(주의) 이 함수는
많은 시간이 소요된다!

'키, 암호문' 쌍을
각각 정수로 파일에 저장한다.

수행 단계를 '.'으로 표시하며
모두 64개가 찍히면 끝난다.





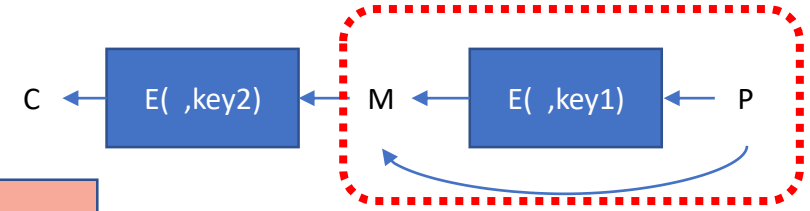

```
pt1 = [0, 1, 2, 3]
make_enc_table(pt1)
(출력)
Encryption Table File
.....................................................................
```

테이블 파일 → 사전 만들기

```
#- 파일에서 테이블을 읽어 사전으로 만든다.
def make_dic_from_enc_table(table_file):
    MaxItems = 1<<24 # 파일의 일부만 읽을때 필요 (최대 2^24까지만)
    dic = {} # (mid, [k1,k2,,])
    f = open(table_file, 'r')
    cnt = 0
    print('Read Encryption Table', end=' ')
    while True:
        line = f.readline().split()
        if not line:
            break
        if cnt > MaxItems :
            break
        cnt += 1

        key1_guess, mid = int(line[0]), int(line[1])
        if mid in  dic :
            dic[mid].append(key1_guess)
        else:
            dic[mid] = [key1_guess]

        if (cnt % (1<<19)) == 0:
            print('.', end='')
    f.close()
    print('\n')

return dic
```

(주의) 고정된 pt에 대하여
mid = E(pt, key) 를 만들면
하나의 mid에 여러 개의 key가 대응될 수 있다.

처음 나오는 mid 값이면 리스트를 만들고, 아니면 기존 리스트에 추가(append)한다.



변수를 파일에 저장하기

- pickle: 변수를 파일로 저장
  - 만드는데 오래 걸리는 변수는 파일에 저장해두고 재사용할 수 있다.
  - 이 과정을 피클링이라고 한다.

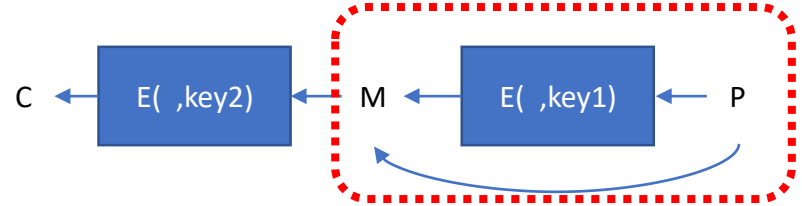
```
import pickle

#--- 변수를 파일에 저장하기
def save_var_to_file(var, filename):
    f = open(filename, 'w+b')
    pickle.dump(var, f)
    f.close()

#--- 파일에서 변수를 가져오기
def load_var_from_file(filename):
    f = open(filename, 'rb')
    var = pickle.load(f)
    f.close()
    return var
```

```
#--- 사전을 파일로 저장(dump)하기
#save_var_to_file(mid_dic, 'TC20Enc_Dic.p')

#-- 파일에서 사전 가져오기
mid_dic2 = load_var_from_file('TC20Enc_Dic.p')
```

(암호문-키) 사전 만들기

```
def make_enc_dic(pt):
    dic = {}
    print('Making Encryption Dictionary', end=' ')
    # 전수조사 범위를 24비트 한정한다 key=[0,*, *, *]
    N = 1<<24

    for idx in range(0, N):
        key = int2list(idx)
        mid = TC20_lib.TC20_Enc(pt, key)
        int_key = list2int(key)
        int_mid = list2int(mid)

        if int_mid in  dic :
            dic[int_mid].append(int_key)
        else:
            dic[int_mid] = [int_key]

        if (idx % (1<<18)) == 0:
            print('.', end='')
    return dic
```

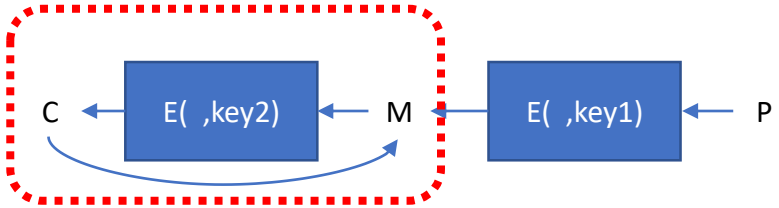
(주의) 이 함수는
많은 시간이 소요된다!



사전에
(key, value) = (암호문, [키리스트])
형식으로 저장한다.

```
pt1 = [0, 1, 2, 3]
mid_dic = make_enc_dic(pt1)
(출력)
Making Encryption Dictionary
.....................................................................
```

MITM-Attack



- MITM(Meet-In-The-Middle) Attack
  - 미리 만든 사전을 이용하여 key2의 가능한 모든 값을 대입한다.
  - key2로 암호문(ct1)을 복호한 결과를 사전에서 찾는다.
  - 사전에 있는 후보키가 새로운 (평문, 암호문)쌍 (pt2, ct2)에 대하여 올바른 결과를 주면 암호키를 찾은 것으로 간주한다.

```
print('Meet in the middle attack', end='')
N = 1<<24  # key2 크기
for idx in range(0, N):
    key2 = int2list(idx)
    mid2 = TC20_lib.TC20_Dec(ct1, key2)
    int_mid2 = list2int(mid2)

    if int_mid2 in mid_dic:
        list_key_candidate = mid_dic[int_mid2]
        if len(list_key_candidate) > 0:
            verify_key_candidate_list(list_key_candidate, key2, pt2, ct2)

    if (idx % (1<<18)) == 0:
        print('.', end='')

print('\n Key search completed!')
```

mid_dic에 (mid, [key list])가 저장되어 있으며, 키 리스트의 길이는 대부분 1로 예상된다. (왜?)

```
(출력)
Meet in the middle attack.
 key1 = [0, 20, 20, 4]  key2 = [0, 1, 2, 3]
.....................................................
 Key search completed!
```

키 후보 리스트

- 키 후보 리스트 탐색
  - 키(key1) 후보를 **정수로** 저장한 리스트에 올바른 키가 있는지 두번째 평문 암호문쌍으로 확인한다.

```
#- 암호키 (공격에서 찾아야 할 키)

def verify_key_candidate_list(key1_list, key2, pt2, ct2):
    flag = False
    for key1 in key1_list:
        key1_state = int2list(key1)

        mid1 = TC20_lib.TC20_Enc(pt2, key1_state)
        ct_comp = TC20_lib.TC20_Enc(mid1, key2)

        if ct_comp == ct2:
            print('\n key1 =', key1_state, ' key2 =', key2)
            flag = True

    return flag
```

(pt2 → ct2)로 암호화하는 경우 올바른 키로 간주하고 출력

```
candidate = [66051, 504503410, 67438087, 3618478663, 1315844, 66051]
verify_key_candidate_list(candidate, key2, pt2, ct2)

(출력)
 key1 = [0, 20, 20, 4]  key2 = [0, 1, 2, 3]
```

Triple DES and Double Enc.

- 1990년대 DES의 암호키(56비트)가 충분한 안전성을 주지 못하게 되어 새로운 대안을 모색
- Double DES(암호키 2개로 두번 암호화)는 안전성을 57비트 밖에 주지 못하게 되어 Triple DES를 대안으로 채택함
  - 두 개의 키를 key1-key2-key1의 순서로 사용하거나
  - 3개의 키를 사용하는 방식이 채택됨
- 현재는 모두 AES로 대체되고, 이전 자료의 복호화에만 권장됨