Cryptanalysis

- 2024-Spring -

Ji, Yong-Hyeon

A document presented for
the Cryptanalysis

Department of Information Security, Cryptology, and Mathematics
College of Science and Technology
Kookmin University

April 20, 2024

Contents

Chapter 1

Midterm

Toy Cipher TC1

TC1Lib.py

```
"""
==========================================
TC1 - Toy Cipher (encryption/decryption)
- n, k: 32-bit
- Identical Round Function
- Key Schedule (X)
==========================================
"""

NUM_ROUND = 10

#--------------------------------------
#  Encryption
#--------------------------------------

#--- S-Box (AES)
Sbox = [ ... ]
ISbox = [ ... ]

#-- AR: Add Roundkey
def AR(in_state, rkey):
   out_state = [0] * len(in_state)
   for i in range(len(in_state)):
      out_state[i] = in_state[i] ^ rkey[i]

   return out_state

#-- SB: Sbox layer
def SB(in_state):
   out_state = [0] * len(in_state)
   for i in range(len(in_state)):
      out_state[i] = Sbox[in_state[i]]
```

```
    return out_state

#-- LM: Linear Map
def LM(in_state):
    out_state = [0] * len(in_state)
    All_Xor = in_state[0] ^ in_state[1] ^ in_state[2] ^ in_state[3]
    for i in range(len(in_state)):
        out_state[i] = All_Xor ^ in_state[i]

    return out_state

#-- Enc_Round
def Enc_Round(in_state, rkey):
    out_state = [0] * len(in_state)
    out_state = AR(in_state, rkey)
    in_state = SB(out_state)
    out_state = LM(in_state)

    return out_state

#- TC1 Encryption
def TC1_Enc(PT, key):
    NROUND = NUM_ROUND # Number of Round = 10
    CT = PT #CT = [0] * len(PT)
    for i in range(NROUND):
        CT = Enc_Round(CT, key)

    return CT

#---------------------------------------
#  Decryption
#---------------------------------------

#-- SB: Sbox layer
def ISB(in_state):
    out_state = [0, 0, 0, 0]
    for i in range(len(in_state)):
        out_state[i] = ISbox[in_state[i]]

    return out_state

#-- Decrypt Round
def Dec_Round(in_state, rkey):
    out_state1 = [0, 0, 0, 0]
    out_state2 = [0, 0, 0, 0]
    out_state3 = [0, 0, 0, 0]
    out_state1 = LM(in_state)
    out_state2 = ISB(out_state1)
```

```
    out_state3 = AR(out_state2, rkey)

    return out_state3

#- TC1 Decryption
def TC1_Dec(input_state, key):
    state = input_state
    numRound = NUM_ROUND # 라운드 수
    for i in range(0, numRound):
        state = Dec_Round(state, key)

    return state

#=======================================
def main():
    message = 'ARIA'
    PT = [ ord(ch) for ch in message ]
    print('Message =', message)
    print('PT =', PT)

    key = [0, 1, 2, 3]
    CT = TC1_Enc(PT, key)
    print('CT =', CT)

    hexPT = [hex(item) for item in PT]
    hexCT = [hex(item) for item in CT]

    print('hexPT =', hexPT)
    print('hexCT =', hexCT)

    bytePT = bytes(PT)
    byteCT = bytes(CT)
    print('bytePT =', bytePT)
    print('byteCT =', byteCT)

    input_state = [202, 134, 119, 230]
    output_state = TC1_Dec(input_state, key)
    print('input ciphertext =', input_state)
    print('output plaintext =', output_state)

if __name__ == '__main__':
    main()
```

```
user@host:~$ python3 TC1Lib.py
Message = ARIA
PT = [65, 82, 73, 65]
CT = [202, 134, 119, 230]
hexPT = ['0x41', '0x52', '0x49', '0x41']
hexCT = ['0xca', '0x86', '0x77', '0xe6']
bytePT = b'ARIA'
```

```
byteCT = b'\xca\x86w\xe6'
input ciphertext = [202, 134, 119, 230]
output plaintext = [65, 82, 73, 65]
```

TC1-TMTO-Table.py

```
#----------------------------------
# TMTO Attack for TC1
# - n: 32-bit, k: 24-bit
# - Parameter: m=t=l=2^8 (mtl = 2^24)
# -- m: number of starting points
# -- l: number of tables
# -- t: length of chain
# - Memory = m*l = 2^16
# - Time   = t*l = 2^16
#----------------------------------

import TC1Lib as TC1
import pickle # store variable
import random # generate random number
import copy   # deep copy

#--- int(4bytes) to list
#--- 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

#--- list to int
#--- [ 0x12, 0x34, 0x56, 0x78 ] -> 0x12345678
def list2int(l):
   n = 0
   num_byte = len(l)
   for i in range(len(l)):
      n += l[i] << 8*(num_byte - i -1)
   return n

#--- Save Variable to File
def save_var_to_file(var, filename):
   f = open(filename, 'w+b')
   pickle.dump(var, f)
   f.close()

#--- Load Variable from File
   def load_var_from_file(filename):
   f = open(filename, 'rb')
   var = pickle.load(f)
   f.close()
   return var
```

```
#==============================================================
#  TMTO Attack
#==============================================================

# 32-bit Enc/Dec : PT = [*,*,*,*] --> CT = [*,*,*,*]
# 24-bit Key     : key = [0,*,*,*]
key_bit = 24

#---------------------------------
# TMTO Table (Dictionary): { (SP:EP) }
#   #SP = #EP = 2^8,   #chains: m = 2^8, #tables: l = 2^8
#---------------------------------

#---------------------------------
# P0 : Chosen Plaintext
# X_{j+1} = E(P0, X_{j})                 # if k = n
# X_{j+1} = R( E(P0, X_{j}) )            # R: 32-bit -> 24-bit
# SP = X0 -> X1 -> X2 -> ... -> Xt = EP  # Encryption Key Chain
#---------------------------------

#-- Reduction FUnction
#-- R: 32-bit [*,*,*,*] -> 24-bit [0,*,*,*]
def R(ct):
   next_key = copy.deepcopy(ct)
   next_key[0] = 0
   return next_key

#-- Create Encryption Key Chain
#-- SP : random key (24-bit)
#-- P0 : chosen plaintext (fixed)
#-- t  : length of chain
def chain_EP(SP, P0, t):
   Xj = SP
   for j in range(0,t):
      ct = TC1.TC1_Enc(P0, Xj)
      Xj = R(ct)   # next Xj (32-bit -> 24-bit)
   return Xj

#--- Debugging Chain
def chain_EP_debug_print(SP, P0, t):
   Xj = SP
   print('SP =', SP)
   for j in range(0,t):
      ct = TC1.TC1_Enc(P0, Xj)
      Xj = R(ct)   # next Xj
      print(' -> ', ct, ' -> ', Xj)
   return Xj
```

```
#--- Debugging Chain
#--- Xj[0,*,*,*] --> ct[*,*,*,*] --> R(ct)[0,*,*,*]
def chain_EP_debug_file(SP, P0, t, chain_num, table_num):
   file_name = 'debug/TMTO-chain-' + str(table_num) + '-' + str(chain_num) +
    ↪  '.txt'
   f = open(file_name, 'w+')
   Xj = SP
   f.write('SP = [0, %d, %d, %d] \n', %(Xj[1], Xj[2], Xj[3]))

   for j in range(0,t):
      ct = TC1.TC1_Enc(P0, Xj)
      Xj = R(ct)
      f.write(' --> [%d, %d, %d, %d] ' %(ct[0], ct[1], ct[2], ct[3]))
      f.write(' --> [%d, %d, %d, %d] \n' %(Xj[0], Xj[1], Xj[2], Xj[3]))
   f.close()

   return Xj

#--------------------------------
# Create one TMTO Table (Number=ell)
# Input:
#      P0: chosen plaintext (fixed)
#       m: number of SPs (row)         m=2^8: SP1 ~ SP2^8
#       t: length of chain (column)    j=0, ... , j=t
#     ell: table number                ell = 0 ~ 255
# Output:
#    dic : { (Key=EP, Value=SP) }
#    path: ./tmto_table/TMTO-ell.dic
#--------------------------------

def make_one_tmto_table(P0, m, t, ell):
   tmto_dic = {}  # (Key, Value) = (EP,SP)
   for i in range(0,m):
      # random starting point
      SP = [0, random.randint(0,255), random.randint(0,255),
       ↪  random.randint(0,255) ]
      EP = chain_EP_debug_file(SP, P0, t, i, ell)
      #EP = chain_EP(SP, P0, t)

      # { (Key=EP, Value=SP) }
      SP_int = list2int(SP)
      EP_int = list2int(EP)
      # EP is Key
      tmto_dic[EP_int] = SP_int

   # files: TMTO-0, TMTO-1, ... , TMTO-255
   file_name = 'tmto_table/TMTO-' + str(ell) + '.dic'
   save_var_to_file(tmto_dic, file_name)
```

```python
#---------------------
# Create total TMTO
# Input:
#   P0: Fixed Plaintext
#   m: no. rows (number of chain)
#   t: no. cols (length of chain)
#   num_of_tables: 2^8 (=256)
#---------------------

def make_all_tmto_tables(P0, m, t, num_of_tables):
   print('making TMTO tables', end='')
   for ell in range(0, num_of_tables):
      make_one_tmto_table(P0, m, t, ell)
      print('.', end='')

   print('\n All TMTO tables are created.')

#=====================
# Test Run

#random.seed(1234)  #fixed seed --> identical result
random.seed(2024)  #fixed seed --> identical result

# Fixed Plaintext
P0 = [1,2,3,4]
# Setup Parameter
m = 256             # m: number of chain
t = 256             # t: length of chain
num_of_tables = 256

#=====================
# (Step 1) Create TMTO Table (Pre-computation)
# TMTO-0, TMTO-1, ...
#=====================
make_all_tmto_tables(P0, m, t, num_of_tables)
```

TC1-TMTO-Attack.py

```python
#-----------------------------------
# TMTO Attack for TC1
# - n: 32-bit, k: 24-bit
# - Parameter: m=t=l=2^8 (mtl = 2^24)
# -- m: number of starting points
# -- l: number of tables
# -- t: length of chain
# - Memory = m*l = 2^16
# - Time   = t*l = 2^16
#-----------------------------------
```

```
import TC1Lib as TC1
import pickle # store variable
import random # generate random number
import copy   # deep copy

#--- int(4bytes) to list
#--- 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

#--- list to int
#--- [ 0x12, 0x34, 0x56, 0x78 ] -> 0x12345678
def list2int(l):
   n = 0
   num_byte = len(l)
   for i in range(len(l)):
      n += l[i] << 8*(num_byte - i -1)

   return n

#--- Save Variable to File
def save_var_to_file(var, filename):
   f = open(filename, 'w+b')
   pickle.dump(var, f)
   f.close()

#--- Load Variable from File
   def load_var_from_file(filename):
   f = open(filename, 'rb')
   var = pickle.load(f)
   f.close()

   return var
#=============================================================
#  TMTO Attack
#=============================================================

# 32-bit Enc/Dec : PT = [*,*,*,*] --> CT = [*,*,*,*]
# 24-bit Key     : key = [0,*,*,*]
key_bit = 24

#--------------------------------
# TMTO Table (Dictionary): { (SP:EP) }
```

```
#   #SP = #EP = 2^8,   #chains: m = 2^8, #tables: l = 2^8
#---------------------------------

#---------------------------------
# P0 : Chosen Plaintext
# X_{j+1} = E(P0, X_{j})                       # if k = n
# X_{j+1} = R( E(P0, X_{j}) )                  # R: 32-bit -> 24-bit
# SP = X0 -> X1 -> X2 -> ... -> Xt = EP  # Encryption Key Chain
#---------------------------------

#-- Reduction FUnction
#-- R: 32-bit [*,*,*,*] -> 24-bit [0,*,*,*]
def R(ct):
   next_key = copy.deepcopy(ct)
   next_key[0] = 0

   return next_key

#-- Create Encryption Key Chain
#-- SP : random key (24-bit)
#-- P0 : chosen plaintext (fixed)
#-- t  : length of chain
def chain_EP(SP, P0, t):
   Xj = SP
   for j in range(0,t):
      ct = TC1.TC1_Enc(P0, Xj)
      Xj = R(ct)   # next Xj (32-bit -> 24-bit)

   return Xj

#--- Debugging Chain
def chain_EP_debug_print(SP, P0, t):
   Xj = SP
   print('SP =', SP)
   for j in range(0,t):
      ct = TC1.TC1_Enc(P0, Xj)
      Xj = R(ct)   # next Xj
      print(' -> ', ct, ' -> ', Xj)

   return Xj

#--- Debugging Chain
def chain_EP_debug_file(SP, P0, t, chain_num, table_num):
   file_name = 'debug/TMTO-chain-' + str(table_num) + '-' + str(chain_num) +
    ↪  '.txt'
   f = open(file_name, 'w+')
   Xj = SP
   f.write('SP = [0, %d, %d, %d] \n', %(Xj[1], Xj[2], Xj[3]))
```

```
    for j in range(0,t):
        ct = TC1.TC1_Enc(P0, Xj)
        Xj = R(ct)
        f.write(' --> [%d, %d, %d, %d] ' %(ct[0], ct[1], ct[2], ct[3]))
        f.write(' --> [%d, %d, %d, %d] \n' %(Xj[0], Xj[1], Xj[2], Xj[3]))
    f.close()

    return Xj

# Chosen Plaintext (Fixed on TMTO Table)
P0 = [1,2,3,4]
# Parameter for Attack
m = 256              # m: Number of Chains over One Table
t = 256              # t: Length of Chain
num_of_tables = 256 # Number of Tables


#======================
# (단계2) 온라인 공격(획득 암호문에 대한 암호키 찾기)
# (단계1에서 만든 사전파일을 이용하여 공격하는 과정)
#======================


#======
# random.seed(2024)으로 만든 TMTO 테이블용 샘플
#======
'''
PTCT for TMTO attack
pt1 = [1, 2, 3, 4]
ct1 = [224, 255, 196, 177]
pt2 = [5, 6, 7, 8]
ct2 = [71, 69, 245, 137]
key = [0, 23, 36, 6]
'''

#--------------
# Key Search for one Table
def one_tmto_table_search(ct, P0, m, t, ell):
    key_candid_list = []
    file_name = 'tmto_table/TMTO-' + str(ell) + '.dic'
    tmto_dic = load_var_from_file(file_name)

    Xj = R(ct)
    current_j = t
    for idx in range(0,t):
        Xj_int = list2int(Xj)

        if Xj_int in tmto_dic: # Xj가 EP에 있는가?
            SP = int2list(tmto_dic[Xj_int]) # dic = { EP:SP }
```

```
            key_guess = chain_EP(SP, P0, current_j - 1)
            key_candid_list.append(key_guess)

        new_ct = TC1.TC1_Enc(P0,Xj)
        Xj = R(new_ct)
        current_j = current_j - 1

    return key_candid_list


#================

ct1 = [224, 255, 196, 177] # (random.seed(2024))
key_pool = []
print("TMTO Attack", end='')
for ell in range(0, num_of_tables):
    key_list = one_tmto_table_search(ct1, P0, m, t, ell)
    key_pool += key_list
    print('.', end='')

print('\n Attack complete!\n')
print('key_pool =', key_pool)

# 다른 (평문, 암호문)을 이용하여, 후보키 중 최종 암호키를 선택함
pt2 = [5,6,7,8]
ct2 = [71, 69, 245, 137] # (random.seed(2024))
final_key = []

for key in key_pool:
    ct_result = TC1.TC1_Enc(pt2, key)
    if ct_result == ct2:
    final_key.append(key)

print('Final key =', final_key)
```

Note (Hellman's Table)**.**

Memory	$m \times l$	m pairs per l tables
Complexity	$t \times l$	t length per l tables

$$SP_1 = X_{1,0} \longrightarrow X_{1,1} \longrightarrow X_{1,2} \longrightarrow \cdots \longrightarrow X_{1,t-1} \longrightarrow X_{1,t} = EP_1$$

$$SP_2 = X_{2,0} \longrightarrow X_{2,1} \longrightarrow X_{2,2} \longrightarrow \cdots \longrightarrow X_{2,t-1} \longrightarrow X_{2,t} = EP_2$$

$$SP_i = X_{i,0} \longrightarrow X_{i,1} \longrightarrow X_{i,2} \longrightarrow \cdots \longrightarrow X_{i,t-1} \longrightarrow X_{i,t} = EP_i$$

$$SP_m = X_{m,0} \longrightarrow X_{m,1} \longrightarrow X_{m,2} \longrightarrow \cdots \longrightarrow X_{m,t-1} \longrightarrow X_{m,t} = EP_m$$

1.1 Time Memory Trade Off (TMTO) Attack

A TMTO attack is typically described in the context of finding the secret key k used in a cryptographic function f. The function f is assumed to be a block cipher or a cryptographic hash function.

Setup

Consider a cryptographic function $f : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$, where $\mathcal{K}$ is the key space, $\mathcal{M}$ is the message space and $\mathcal{C}$ is the cipher space. The goal is to invert f given $f(k)$, i.e., to find k when $f(k)$ is known.

Precomputation Phase

In the precomputation phase, a series of computations are performed to create a trade-off between the computation time and memory usage:

1. Select a subset of keys $\{k_1, k_2, \ldots, k_t\} \subset \mathcal{K}$.
2. Compute $f(k_i)$ for each k_i.
3. Store the pairs $(k_i, f(k_i))$ in a table called the **precomputed table**.

This table is used to accelerate the recovery of k by storing potential outputs and their corresponding inputs.

Recovery Phase

Given a ciphertext c, the attacker attempts to find k such that $f(k) = c$:

1. For each potential key k', compute $f(k')$.
2. Check if $f(k')$ exists in the precomputed table.
3. If a match is found, i.e., $f(k') = f(k_i)$ for some i, retrieve k_i.

Complexity Analysis

The effectiveness of a TMTO attack depends on the sizes of the key space $\mathcal{K}$, the cipher space C, and the table:

- **Memory Requirement:** Proportional to the number of entries t in the table.
- **Time Complexity:** Proportional to $\frac{|\mathcal{K}|}{t}$, assuming uniform distribution and independent choices of k_i.

Example: Hellman's TMTO

Hellman's approach involves structuring the precomputed table in chains where each chain starts from a randomly chosen initial value k_0 and is constructed as follows:

$$\begin{aligned} k_1 &= f(k_0), \\ k_2 &= f(f(k_0)), \\ &\vdots \\ k_t &= f^{(t)}(k_0), \end{aligned}$$

where $f^{(t)}$ denotes the t-th application of f. Only k_0 and k_t are stored, reducing memory usage but requiring more time in the recovery phase to reconstruct chains.

Introduction

Hellman's Time-Memory Trade-Off is a cryptographic attack method that uses precomputed tables to find key inverses of encryption functions more quickly than by brute force alone. This technique involves creating a series of tables that map encrypted values to potential keys.

Hellman Tables

Let $f : \mathcal{K} \to C$ be a cryptographic function, where $\mathcal{K}$ represents the key space and C represents the ciphertext space. The function f is typically a block cipher encryption under a specific key.

Table Structure

A Hellman table is built to store chains of keys and ciphertexts that are produced by repeatedly applying f. The process to create a single table is as follows:

1. **Initialization:** Choose m initial values $k_0^{(1)}, k_0^{(2)}, \ldots, k_0^{(m)} \in \mathcal{K}$.

2. **Chain Generation:** For each initial value $k_0^{(i)}$, generate a chain of length t:

$$
\begin{aligned}
k_1^{(i)} &= f(k_0^{(i)}),\\
k_2^{(i)} &= f(k_1^{(i)}),\\
&\vdots\\
k_t^{(i)} &= f(k_{t-1}^{(i)}).
\end{aligned}
$$

3. **Storing Endpoints:** Store only the starting point $k_0^{(i)}$ and the endpoint $k_t^{(i)}$ for each chain.

The table thus contains m entries of the form $(k_0^{(i)}, k_t^{(i)})$. Multiple tables can be constructed to cover more possible keys.

Recovery Phase

To recover the key k given a ciphertext c, the following steps are performed:

1. **Chain Traversal:** For each endpoint $(k_0^{(i)}, k_t^{(i)})$ in the table:

   (a) Compute backwards from c, applying f^{-1} if possible, or guess intermediate values to trace back the chain.

   (b) If an intermediate value $k_j^{(i)}$ matches the ciphertext c, the corresponding starting point $k_0^{(i)}$ is used to regenerate the chain forward to find the preimage of c.

2. **Key Verification:** Verify by applying f on the found preimage to check if it indeed maps to c.

Complexity and Efficiency

The effectiveness and efficiency of Hellman tables are characterized by:

- **Memory Usage:** Proportional to the number of chains m.
- **Time Complexity:** Proportional to the product of the number of tables and the length of the chains t, divided by the number of chains m.
- **Trade-Off:** By adjusting m and t, a trade-off between precomputation time, memory usage, and recovery speed can be achieved.

Conclusion

Hellman's TMTO using Hellman Tables provides a method to potentially reduce the complexity of reversing cryptographic functions from $O(|\mathcal{K}|)$ to $O(\sqrt{|\mathcal{K}|})$ under ideal conditions, making it a powerful tool in cryptographic attacks.