Hardware Security

Lab 1:

Timing attack against a DES software implementation

Marouene Boubakri

marouene.boubakri@eurecom.fr

Contents

Introduction 1
Timing Attack 2
Timing attack against a DES software implementation 4
Attack Advancement 7
Fixing the permutation function 7

Introduction

In this report I will try to explain how an attacker can exploit a flaw in a DES software implementation which computation time depends on the input messages and on the secret key.

The most basic attack technique against ciphers is brute force (trying all possible keys). The complexity and the feasibility of this method are determined by the key length.

Data Encryption Standard (DES) is a symmetric-key algorithm (the same key is used for encryption/decryption).

Many research results demonstrate that the algorithm has theoretical weaknesses. It is considered as insecure due to the key size which is 56 bits. To break it an attacker can try 2^56 possible keys until he finds the correct decryption key($200 machine with minimal computing power takes only few days to find the key).

Timing Attack

DES implementation in applications may present flaws also. There are other approaches which exploit these flaws to find the key with less complexity. One known method is Timing Attack which is widely used in cryptanalysis.

I will show through a typical example how can I extract the final round key used to encrypt data by exploiting DES implementation weakness.

Assume that the following pseudo-code is a part of an encryption algorithm implementation:

```
1.     s = p xor k;
2.     foreach bit b in s
3.     {
4.      if(b == 1)
5.        calculation();
6.     }
7.     endforeach;
```

A variable **s** stores the result of an xor operation between an input **p** and a key **k**. The size of p and k is 4 bits. Then I extract each bit from s and test if it is equal to 1. If so, a calculation routine is executed. Assume that the calculation function takes 1 ms.

Now an attacker wants to find the key k used during the operation. The attacker knows the implementation so he knows that the execution time depends on the number of bits equal to 1 in s. He builds a measurement table containing the execution time for each input p.

p	exectime
0001	2 ms
0010	4 ms
...	...
1111	2 ms

For a random key k and input p the attacker can predict how much time the execution would take.He can find the key which corresponds to the real measurement by trying all possible values.

So who can the attacker find the key?

1- First he generates a random key in (0..15)
2- Then he compute s = p xor k for all possible p values
3- He computes the numbers of bits for each s
4- Finally for each key guess compare the predicted numbers of bits with the measured time of the execution with unknown key

p	k=0000	k=0001	k=...	k=1101	exectime
0001	1	1	...	2	2 ms
0010	1	2	…	4	4 ms
…	…	…	…	…	…
1111	1	2	…	2	2 ms

To automate the comparison in step4 statistical tools are used such as Pearson Correlation Coefficient which is a measure of the linear correlation between 2 variables x and y.

if x and y have a relationship (y = ax + b) then the coefficient = 1. (a and b are constants).

In practice the measurements may not be perfect because the execution might perform additional time variant information.

But according to Law of large numbers (LLN) the average of the results obtained from a large number of trials should be close to the expected value, and will tend to become closer as more trials are performed.

So the attacker should perform a lot of executions for the same input p and average them all to tend towards a constant value.

Timing attack against a DES software implementation

Now I will explain timing attack but now with DES softwareimplementation.

The P permutation routine is implemented as follow:

```
1.      uint64_t des_p_ta(uint64_t val) {
2.        uint64_t res;
3.        inti, j, k;
4.
5.        res = UINT64_C(0);
6.        k = 0;
7.        for(i = 1; i<= 32; i++) {
8.          if(get_bit(i, val) == 1) {
9.            for(j = 1; j <= 32; j++) {
10.             if(p_table[j - 1] == i) {
11.               k = j;
12.             }
13.           }
14.           res = set_bit(k, res);
15.         }
16.       }
17.       return res;
18.     }
```

In the above code, lines from 9 to 14 are executed only if the current bit of val is equal to 1 (line 8).

This is similar to the first example I provided.

Since I have control over the algorithm implementation I can perform a lot of acquisitions and store them in a file. An acquisition is a cipher text with corresponding execution time.

Following is the keys used during acquisitions:

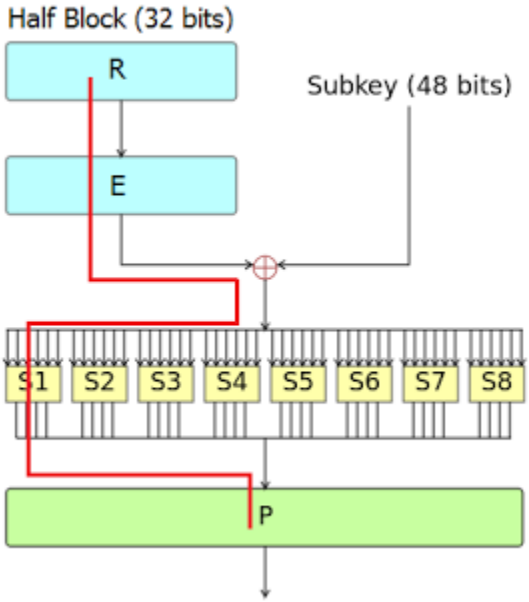
64-bits key (with parity bits): 0xdeadbeefcafed0dd
56-bits key (without parity bits): 0xfff92ee3dafbf5
48-bits round key 1 - 6-bits subkeys: 0x3ff3d5afbc7d - 0x0f 0x3f 0x0f 0x15 …
…
48-bits round key 16 - 6-bits subkeys: 0x2e7ffbf8ff9d - 0x0b 0x27 0x3f 0x3b …

The final round key is 0x2e7ffbf8ff9d

Following is a part of the acquisition file:

0x67b29699175cb7f7 178161.600000
0xe267bca35d392fe7 170923.600000
0xba0e8cd2e5fbfd33 177774.000000

The following figure illustrates the attack path (in red) performed in a DES round:



R is the right half of the cipher text. R is expanded by E function then xored with the subkey(unknown). Then for each 6 bits of E(R) I get 4bits from the Sbox.

I know the cipher text so I know R. E and P are also known. As I mentioned the execution time depends on the number of input bits of the P function.

Assume that INP is the input of P function.

INP = sbox[subkey xor E(R)]

The sboxes are independent so for each 4 bits I have:

INP1_4 = sbox1[subkey1_6 xor E(R)1_6]

INP5_9 = sbox2[subkey7_13 xor E(R)7_13]

..

..

The attack is performed in each 4 bits separately. For the first 4 bits:

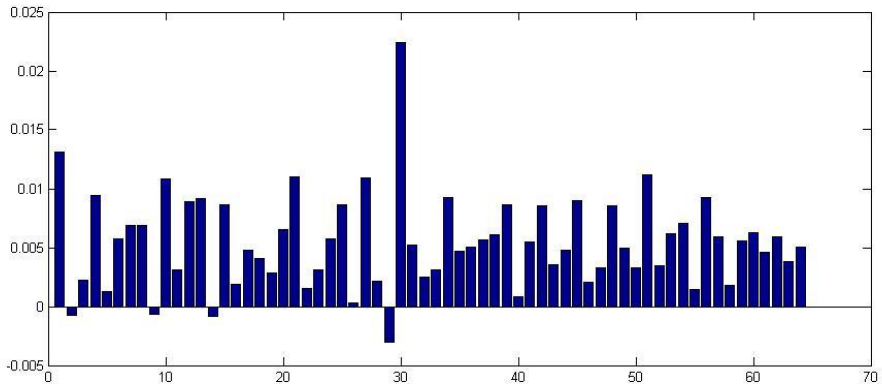
The time model is the number of bits = 1 in INP1_4

HW(INP1_4) is the hamming weight of INP1_4 which is the time model.

The final expression is the following:

HW(INP1_4) = HW(sbox1[subkey1_6 xor E(R)1_6])

This expression is computed for each subkey1_6 (the first 6 bits of the unknown subkey) and all the acquisitions I have. This gives 64 time models for each subkey value. These models are correlated with the real measurements (using PCC).As a result only the correlation for one subkey is the largest value between the other 63 models. See the figure below.



Like this I get the first 6 bits of the unknown subkey and to get the subkey I compute time models for all remaining sbox outputs.

Following is a part of the timing attack script:

```
1.      rk = bk = 0x000000000000
2.          delta = 0
3.          for sbox in reversed(xrange(8)):
4.              mask = 63 << (42 - 6*sbox)
5.              rk &= ~mask
6.              for i in range(64):
7.                      key =  i << (42 - 6*sbox)
8.                      ctx = pcc.pccContext (1)
9.                      for j in range(args.n):
10.                             r16l16 = des.ip (ct[j])
```

```
                        l16 = des.right_half (r16l16)
                        sbo = des.sboxes (des.e (l16) ^ (rk | key))
                        hw = hamming_weight (sbo)
                        ctx.insert_x(t[j])
                        ctx.insert_y(0, hw)
                ctx.consolidate ()
                new_delta = ctx.get_pcc(0)
                if new_delta > delta:
                        delta = new_delta
                        bk = key
        delta = 0
        rk |= bk
    print "Average timing: %f" % (sum (t) / args.n)
    print "Last round key (hex):"
    print ("0x%012X" % rk)
```

Following is the output generated by my timing attack script:

root@maro-vm:~/eurecom/hwsec/lab# python ta.py ta.dat 3700
Average timing: 134301.748360
Last round key (hex):
0x2E7FFBF8FF9D

The script found the correct final round key.

Attack Advancement

My attack script successfully retrieved the final round key using 3700 guesses. There are advancements that can decrease the number of required acquisition. One can think about making a time model for 2 S-Boxes output.

Fixing the permutation function

To fix the implementation weakness I propose the following implementation for the permutation function.

```
uint64_t des_p_ta (uint64_t arg)
{
    uint64_t res = 0;
    int i, val, pos;
    for (i = 1; i <= 32; i++) {
        pos = p_table[i-1];
        val = get_bit(pos, arg);
        res = force_bit(i, val, res);
```

```
        }
        return res;
    }
```

In the above implementation I removed conditional jumps to make the execution time constant. After compiling the DES sources with this fix I was unable to find the last round key for 10000 acquisitions.

Following is the disassembly of the des_p_ta() function:

```
Disassembly of section .text:

0000000000000000 <des_p_ta>:
   0:   53                      push   %rbx
   1:   41 b8 00 00 00 00       mov    $0x0,%r8d
   7:   ba 1f 00 00 00          mov    $0x1f,%edx
   c:   31 c0                   xor    %eax,%eax
   e:   bb 20 00 00 00          mov    $0x20,%ebx
  13:   41 bb 01 00 00 00       mov    $0x1,%r11d
  19:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)
  20:   89 d9                   mov    %ebx,%ecx
  22:   41 2b 08                sub    (%r8),%ecx
  25:   49 89 fa                mov    %rdi,%r10
  28:   4d 89 d9                mov    %r11,%r9
  2b:   49 83 c0 04             add    $0x4,%r8
  2f:   49 d3 ea                shr    %cl,%r10
  32:   89 d1                   mov    %edx,%ecx
  34:   83 ea 01                sub    $0x1,%edx
  37:   49 d3 e1                shl    %cl,%r9
  3a:   4c 89 ce                mov    %r9,%rsi
  3d:   48 f7 d6                not    %rsi
  40:   48 21 c6                and    %rax,%rsi
  43:   4c 89 d0                mov    %r10,%rax
  46:   83 e0 01                and    $0x1,%eax
  49:   48 d3 e0                shl    %cl,%rax
  4c:   4c 21 c8                and    %r9,%rax
  4f:   48 09 f0                or     %rsi,%rax
  52:   83 fa ff                cmp    $0xffffffff,%edx
  55:   75 c9                   jne    20 <des_p_ta+0x20>
  57:   5b                      pop    %rbx
  58:   c3                      retq
  59:   0f 1f 80 00 00 00 00    nopl   0x0(%rax)

0000000000000060 <get_bit>:
  60:   b9 20 00 00 00          mov    $0x20,%ecx
  65:   29 f9                   sub    %edi,%ecx
  67:   48 d3 ee                shr    %cl,%rsi
  6a:   83 e6 01                and    $0x1,%esi
  6d:   89 f0                   mov    %esi,%eax
  6f:   c3                      retq
```

```
42.     0000000000000070 <force_bit>:
43.       70:   b9 20 00 00 00          mov    $0x20,%ecx
44.       75:   48 63 c6                movslq %esi,%rax
45.       78:   29 f9                   sub    %edi,%ecx
46.       7a:   bf 01 00 00 00          mov    $0x1,%edi
47.       7f:   48 d3 e7                shl    %cl,%rdi
48.       82:   48 d3 e0                shl    %cl,%rax
49.       85:   48 21 f8                and    %rdi,%rax
50.       88:   48 f7 d7                not    %rdi
51.       8b:   48 21 d7                and    %rdx,%rdi
52.       8e:   48 09 f8                or     %rdi,%rax
53.       91:   c3                      retq
```

As the disassembly shows there are no conditional jumps except the end of loop jump. I conclude that the implementation is safe now.