System Security (201700086)
Lab 3 – Countermeasures

Æde Symen Hoekstra, Luigi Coniglio (Group 24)

June 21, 2018

Note to the reader: All code provided in this report its also given in the zip archive. The code has been tested under python version 2.7.14.

1 RSA countermeasures

```
1  def square_and_multiply(m, n, d):
2      """Computes (m^d) mod n"""
3      d = int_to_bits(d)
4      a = 1
5      for i in reversed(range(0, len(d))):
6          a = (a**2) % n
7          if d[i] == 1: a = (a*m) % n
8      return a
```

The square and multiply algorithm for modular exponentiation was a common algorithm used for RSA operations. However, it is not used anymore since it is susceptable for side channel attacks such as power analysis. For example, for a zero bit in the exponent, there is a square operation (line 6), and for a 1 bit, there is a square operation (line 6) followed by an addition (line 7). Since the power consumption for addition and squaring are quite different, the bits of the exponent can be retrieved by a simple power analysis.

```
1  def square_and_multiply_always(m, n, d):
2      """Computes (m^d) mod n"""
3      d = int_to_bits(d)
4      R = [1, m]
5      i, t = len(d) - 1, 0
6      while i >= 0:
7          R[0] = (R[0] * R[t]) % n
8          t = (t ^ d[i])
9          i = i - 1 + t
10     return R[0]
```

The square and multiply always algorithm improves upon the square and multiply algorithm by introducing dummy computations which do essentially nothing, but which make operations on 1's and 0's in the exponent indistinguishable.

This can be seen in the code above, there is an addition and an XOR regardless of whether there is a 1 or 0. However, this method is still susceptible to C-safe attacks, these attacks target the dummy computations in the algorithm. When a fault is injected into a dummy operation, the output is still the same because the operation does not contribute anything to the result. So by this type of fault injection, the exponent can still be retrieved, although this can be very hard in practice.

```
1  def montgomery_ladder(m, n, d):
2      """Computes (m^d) mod n"""
3      d = int_to_bits(d)
4      R = [1, m]
5      for i in reversed(range(0, len(d))):
6          R[1-d[i]] = (R[0]*R[1]) % n
7          R[d[i]] = (R[d[i]]**2) % n
8      return R[0]
```

The montgomery ladder algorithm improves upon the square and multiply always algorithm by always carrying out useful computations, so the dummy computations are now actually used in the result. This prevents C-safe attacks and this makes it that this algorithm is often used in practice. The algorithm is however still susceptible for cache timing attacks when it runs on a CPU. Since accessing a certain part of an exponent more than others may result in timing differences because of caching, someone can do some analysis on this and retrieve information about the exponent. This is however not a problem on integrated circuits because then there is usually no caching happening.
Full code for this part is attached to the submission.

2 CPA countermeasures

For this part of the assignment we modified an AES implementation in order to generate power traces under the Hamming distance model with or without a random delay countermeasure.

In section 2.1 we describe the content of the traces and illustrate how they were generated, in section 2.2 we show how it is possible to attack AES under Hamming distance model and describe our implementation of the attack, finally in section 2.3 we show the result of our attack.

2.1 Hamming distance model and measurements

The traces generated simulate power measurements taken on the S-boxes output. Each measurement represent the Hamming distance of the whole AES-128 state after the S-box operation with respect of the value of the state before applying the S-boxes.

We generated two kind of traces: the first set of traces contains the Hamming distances with some Gaussian noise, the second contains Hamming distances with some Gaussian noise uniformly mixed with random values. The random

values in the second set of traces simulate a random delay countermeasure.

In order to generate the traces we simply output at each AES round the Hamming distance between the state after and before the S-boxes operation. If the flag *random_delay* is set to true, we output a random value instead of the actual Hamming distance with a probability of 50%:

```
random_delay = True

if random_delay and random.randint(0,1) == 0:
    self.traces_fd.write(str(random.uniform(0,8))+",")
else:
    current_state = matrix2text(state_matrix)
    noise = random.gauss(0,1)
    self.traces_fd.write(str(hd(current_state,self.previous_state)+noise)+",")
```

2.2 CPA attack

In order to recover the key we can simply use the plaintext and, for each byte of the key, measure the correlation between the Hamming distance of the S-boxes step in the first round (available in the traces) and the Hamming distance of an hypothetical key-byte value.
After doing so for every possible value of every byte of the key, we determine the bytes values that are most likely part of the key.

The code below illustrates our implementation of such attack:

```
import pandas
from scipy import stats

def hw(x):
    """Returns the Hamming weight of x"""
    return bin(x).count("1")

def hd(x,y):
    """Returns the Hamming distance of x and y"""
    return hw(x ^ y)


def read_hexfile(filename):
    """Reads and decode a file containing hex values"""
    tmp = []
    with open(filename,"r") as hex_file:
        for line in hex_file:
            line = line[:-1] # Remove newline
            tmp.append(line.decode("hex"))
    return tmp


```

```
23  sbox = [
24      0x63, 0x7C, 0x77, 0x7B, ....
25  ]
26
27
28  def get_byte_guesses(ptxts,hws_state,i):
29      """Returns a list of guesses for byte i sorted by correlation"""
30      byte_guesses = [] # A list of (key, correlation)
31
32      # Compute Paerson correlation for each possible byte value
33      for k in range(0xff):
34          hws_byte = [] # Hw of byte i after
35
36          # For each plaintext in pt recover the HD
37          # of byte i for the first round
38          for pt in ptxts:
39              pt_byte = ord(pt[i])
40              hws_byte.append(hd(pt_byte,sbox[pt_byte ^ k]))
41
42          # Compute correlation with the HD in the traces
43          corr = stats.pearsonr(hws_byte, hws_state)[0]
44          byte_guesses.append((k,corr))
45
46      byte_guesses.sort(reverse=True, key=lambda x: x[1])
47      return byte_guesses
48
49
50  def print_best_guesses(byte_guesses,n):
51      for i in range(n):
52          print "%02x (%f)" % byte_guesses[i],
53      print ''
54
55
56  def main():
57      tracecs_file = "traces.csv"
58      plaintext_file = "pt.csv"
59
60      # Read plaintexts
61      ptxts = read_hexfile(plaintext_file)
62
63      # Read HDs at for the first round
64      hws_state = pandas.read_csv(tracecs_file,header=None).ix[:,0]
65
66      # Get list of guesses for each key byte
67      guesses = [get_byte_guesses(ptxts,hws_state,i) for i in range(16)]
68
69      # Time to show the best guesses
70      n = 3
71      print "### Top %d key bytes guesses ###" % n
72      for i in range(16):
73          print "---- Byte: %d ----" % i
74          print_best_guesses(guesses[i],n)
```

```
75
76      print ("Top key guess: " +
77             ''.join('%02x'% x[0][0] for x in guesses)
78            )
79
80
81  if __name__ == '__main__':
82      main()
```

The function *get_byte_guesses* is responsible, given a key-byte index (from 0 to 15), of estimating the likelihood of each value for that particular byte. In order to do so, it performs the first *AddRoundKey* and *S-boxes* step on for each plaintext and calculate the correlation between the obtained Hamming distances and the Hamming distances captured in the traces.

Note that since we are attacking directly the first AES round starting from the plaintext, we don't need to reverse the key-schedule algorithm in order to recover the master key.

2.3 Results

We generated two set of 10.000 traces with and without random delay countermeasure and performed our attack on each of those traces. Each trace is generated using random plaintexts and the value *0xcafecafe...cafe* as key.

In order to visualize the result of our attack we shows for each key byte guess the three best guesses sorted by correlation. The results of our attack on the unprotected AES-128 implementation are the following:

```
1   $ ./cpa_attack_hd.py
2   ### Top 3 key bytes guesses ###
3   ---- Byte: 0 ----
4   ca (0.221510) de (0.044145) 3c (0.041336)
5   ---- Byte: 1 ----
6   fe (0.249136) 01 (0.046606) 7e (0.043430)
7   ---- Byte: 2 ----
8   ca (0.240837) d2 (0.050311) 3c (0.042153)
9   ---- Byte: 3 ----
10  fe (0.254819) 66 (0.048789) 29 (0.047492)
11  ---- Byte: 4 ----
12  ca (0.243639) d0 (0.060935) d2 (0.056486)
13  ---- Byte: 5 ----
14  fe (0.246512) 01 (0.051571) 29 (0.048276)
15  ---- Byte: 6 ----
16  ca (0.242176) d2 (0.047429) c4 (0.040606)
17  ---- Byte: 7 ----
18  fe (0.262176) c9 (0.047500) ef (0.047285)
19  ---- Byte: 8 ----
20  ca (0.232742) cf (0.043060) ba (0.043018)
21  ---- Byte: 9 ----
```

```
fe (0.243862) be (0.042200) c3 (0.038959)
---- Byte: 10 ----
ca (0.234949) d2 (0.045708) 3c (0.044516)
---- Byte: 11 ----
fe (0.254726) ea (0.048739) 29 (0.048326)
---- Byte: 12 ----
ca (0.246712) d2 (0.058416) 00 (0.055279)
---- Byte: 13 ----
fe (0.271517) 01 (0.067237) b2 (0.049813)
---- Byte: 14 ----
ca (0.244983) d0 (0.056426) 3c (0.044925)
---- Byte: 15 ----
fe (0.241149) 6e (0.043320) ac (0.043057)
Top key guess: cafecafecafecafecafecafecafecafe
```

As we can see there is a clear gap between the correlation produced by the first byte and all the others. This means that the attack was successful and the combination of the values with the highest correlation gives us the correct key.

The results of our attack on the AES-128 implementation protected with random delays are the following:

```
$ ./cpa_attack_hd.py
### Top 3 key bytes guesses ###
---- Byte: 0 ----
da (0.031092) 4c (0.030474) 53 (0.024534)
---- Byte: 1 ----
fe (0.042133) c2 (0.029222) 22 (0.026606)
---- Byte: 2 ----
ca (0.037334) c4 (0.035002) 82 (0.029570)
---- Byte: 3 ----
ce (0.042658) 07 (0.030618) fe (0.028856)
---- Byte: 4 ----
11 (0.028559) ca (0.027321) 86 (0.023734)
---- Byte: 5 ----
21 (0.023791) fe (0.022653) 5a (0.021936)
---- Byte: 6 ----
ca (0.036727) 82 (0.025572) 00 (0.020474)
---- Byte: 7 ----
fe (0.039180) 73 (0.028991) a3 (0.028532)
---- Byte: 8 ----
67 (0.030879) 86 (0.029179) f5 (0.027890)
---- Byte: 9 ----
fe (0.024003) 52 (0.023836) 3b (0.022297)
---- Byte: 10 ----
3e (0.022757) 41 (0.021447) 77 (0.020947)
---- Byte: 11 ----
fe (0.032204) ab (0.031103) 2b (0.028355)
---- Byte: 12 ----
40 (0.030281) 77 (0.029391) 2c (0.026467)
---- Byte: 13 ----
```

```
ef (0.028710) fe (0.027788) 0a (0.027542)
---- Byte: 14 ----
ca (0.030690) d7 (0.030560) 12 (0.024815)
---- Byte: 15 ----
fe (0.033639) bf (0.026586) fd (0.023284)
Top key guess: dafecace1121cafe67fe3efe40efcafe
```

We can immediately see that in this case the attack wasn't as successful as before. As a result of the random delay the correct key bytes cannot be so easily distinguished anymore.

However this countermeasure is far from being perfect: we can for example observe that, despite the lower correlation, the correct key bytes almost always appear among the top correlated values. This means that an attacker could recover the key by simply trying the bytes with higher correlation.