Secure Hardware and Embedded Devices

(ELEC6237)

Coursework Answer Sheet

Please write your name and student number below:

Name: _____ _____

Student ID: _____

Section A

Task 1

- Change the input vector to IN = [1] and execute the code again. Explain any differences between your results and the one shown above

  The number of dimensions in input vector is equal to row number in both matrixs of "excTime" and "predTime". The output matches the first row in original output, which is the corresponding result of "input =1" in original input. The rest part in original output disappear for no corresponding input.

- Will you be able to guess the correct key if you use the input vector in step 2? Explain your answer.

  Unable to guess the correct key if only one input. A single input may have 5 different Hamming Weight(0,1,2,3,4). Unless this single input meet correct key occasionally, it would have a combination of possible key through same execution time. It is impossible to judge the correct key from combination if there is no additional evidence.

Task 2

- Vary the value of N in the code, then estimate the minimum value of N, needed to get an accurate estimation of the execution time.

  From a view of probability and statistics, since utilization of 'The law of large numbers' has worked, it is appropriate to consider this case as a Normal Approximation to the Binomial Distribution. The approximation is good for $np > 5$ *and* $n(1-p) > 5$*, in which* n *is sample size and* p *is the*

proportion of success in the population. For a binomial variable X, $E(X) = np$ and $V(X) = np(1-p)$.

In this case, because the average of Hamming weight is 2 and total input bits is 4, it is appropriate to make $p = 0.5$. Here the setting of confident value is 95%. Therefore, the $\alpha = 0.05$ and $z_{\alpha/2} = z_{0.025} = 1.96$.

$$n = \left(\frac{z_{\alpha/2}}{E}\right)^2 p(1-p) \qquad \text{Equation 1.1}^{[1]}$$

Therefore, estimation of the minimum value of N could be conclude through Equation 1.1, in which $E = |p - \hat{P}|$. $\hat{P}$ is the point estimator of p. In this case, we could consider $E = \left|\frac{predicted\ execution\ time - practical\ execution\ time}{average\ execution\ time}\right|$.

The difference between the predicted and actual execution times could be considered as the acceptable error range. Therefore, we could conclude following Table 1 to estimate the minimum value of N.

Table 1 the minimum value of N and the error range

error range	*0.05*	*0.10*	*0.15*	*0.20*	*0.25*	*0.30*	*0.35*	*0.40*	*0.45*	*0.50*
N	24586.2	6146.5	2731.8	1536.6	983.4	682.9	501.7	384.1	303.5	245.8

As a result, if we round practical execution time to the nearest integer number, which means error range is 0.50, the minimum value of N = 246, the output is placed in Figure 1.2.1.

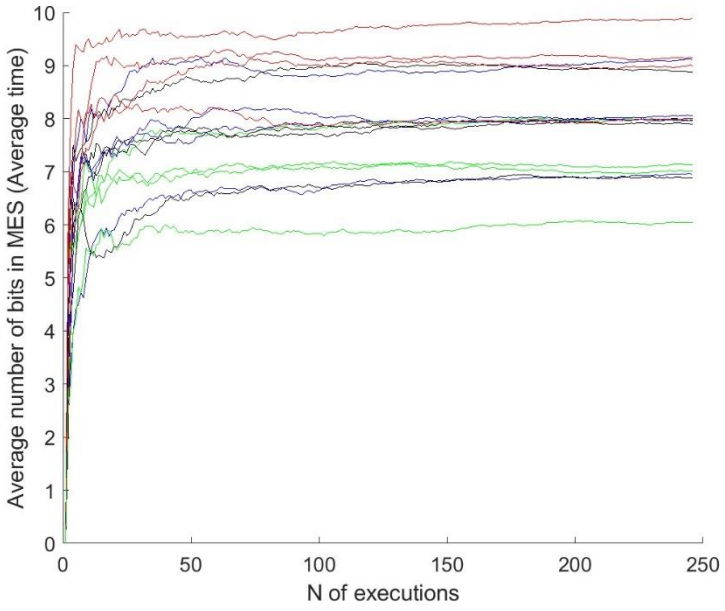


Figure 1.2.1 output of N = 246

- Explain the significance of the minimum value of N, in the context of a practical side-channel attack.

  The evaluation of minimum N would indicate the least operation times for attack. 'N' refers to sampling times. In side-channel attack, it present that how many sample Hackers need to find the correct key. The minimum value of N means the least sample to get accurate result, which shows efficiency of attack.

Task 3

- This example uses Pearson Correlation Coefficient, what other statistical metrics can be used in this case? Justify your answer with experimental evidence.

  Spearman Coefficient and Kendall Coefficient. The outputs have shown in Figure 1.3.1, which is correct result for input"6CE3". The size of these three Coefficients in practical operation also show in Figure 1.3.2, with code in Figure 1.3.3.

```
guessedKeyNibble1 =

     3


guessedKeyNibble2 =

     3


guessedKeyNibble3 =

     3
```

Figure 1.3.1 output of three coefficients

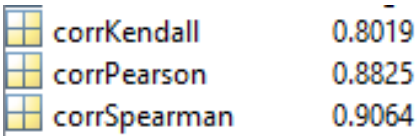


Figure 1.3.2 the size of three coefficient

```
Testmatrix1 =corr(timeModel,'Type', 'Pearson');
Testmatrix1;
Rp = Testmatrix1(1,2:17);
[corrPearson,idx1] = max(Rp);
guessedKeyNibble1 = idx1-1;
guessedKeyNibble1

Testmatrix2 =corr(timeModel,'Type', 'Spearman');
Testmatrix2;
Rs = Testmatrix2(1,2:17);
[corrSpearman, idx2] = max(Rs);
guessedKeyNibble2= idx2-1;
guessedKeyNibble2

Testmatrix3 =corr(timeModel,'Type', 'Kendall');
Testmatrix3;
Rk = Testmatrix3(1,2:17);
[corrKendall, idx3] = max(Rk);
guessedKeyNibble3= idx3-1;
guessedKeyNibble3
```

Figure 1.3.3 changed code for three coefficents

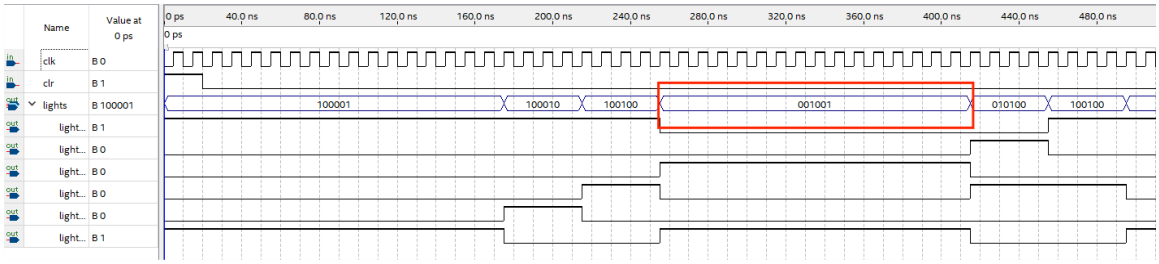
Section B

Answer ONE of the following two questions:

Question B-1

- Explain using experimental supportive evidence whether or not the design (v1) has a hardware Trojan.

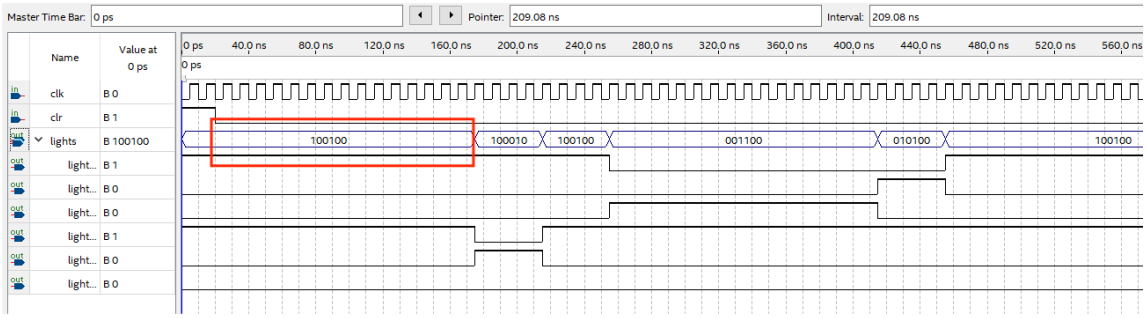
  Yes, it has. The output of 'S3' is '001001', which should be '001100' in original design.



- Explain using experimental supportive evidence whether or not the design (v2) has a hardware Trojan.

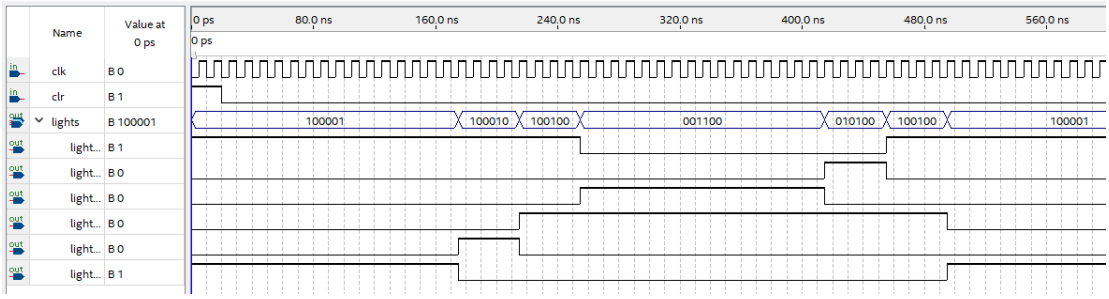
  Yes, it has. The output of 'S0' is '100100', which should be '100001' in original design. It seems that 'S5' disappeared. Acctually, because the designed output of 'S5' is also '100100', the 'S5' in previous round has combined with 'S0' in next

round. Therefore, it forms a output of '100100' in 19 clock cycles(adding 1 clock cycles to 15+3 for signal transform).



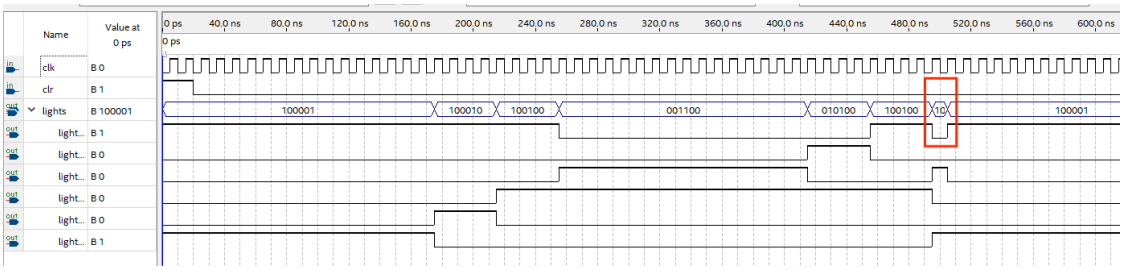
- Explain using experimental supportive evidence whether or not the design (v3) has a hardware Trojan.

  NO, the outputs are same to designed outputs.



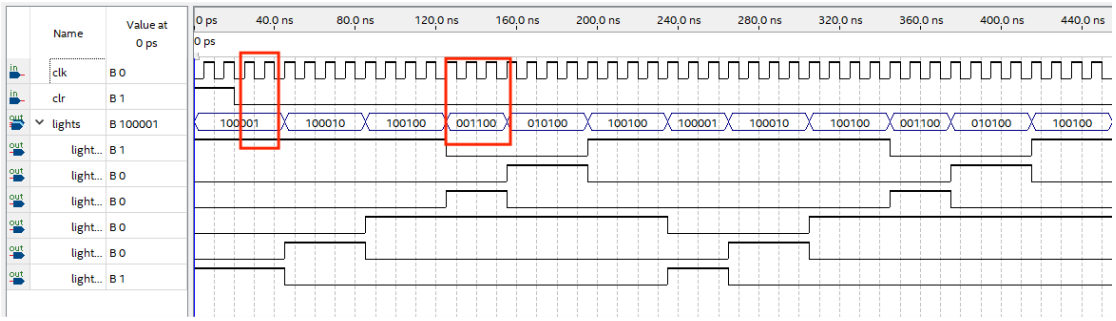
- Explain using experimental supportive evidence whether or not the design (v4) has a hardware Trojan.

  Yes, between 'S5' in previous round and 'S0' in next round, there exists an additional state that is not designed.



- Explain using experimental supportive evidence whether or not the design (v5) has a hardware Trojan.

  Yes, the outputs indicate that this design stays in 'S0' and 'S3' only for 2 clock cycles, which should be 15 clock cycles.



Question B-2

- Draw a Data Flow Diagram, using the suggestions and guidance from the lecture notes, that shows the system on a level that is appropriate for architectural threat analysis

  (Insert your answer here)

- Identify ten assets in the system, and justify your choice in each case

  (Insert your answer here)

- For each asset you have chosen, identify one security threat based on the STRIDE model, explaining how such threat might materialize (i.e. give an example of a relevant security attack technique)

  (Insert your answer here)

- For each identified security threat, provide a countermeasure

  (Insert your answer here)

Section C

- Explain how you designed your algorithm to meet the specifications with appropriate supporting evidence to demonstrate correct functionality.

  *Above all, the entire design consists of 5 specific parts: (1)transform between 16-bits and a 2*2 matrix with 4 nibbles, (2)substitutes each input nibble, (3)shift row, (4) MixColumn, (5)generation of key-schedule and key-addition.*

  *(1) transform between 16-bits and a 2*2 matrix*
  *To make operation in Encryption visualized as theory, the 16-bits input should transform into a 2*2 matrix.*
  First, divide 16-bits into 4 nibbles through Bitwise Operators. Each nibble is extract through bitwise shift and AND operation. Then, utilize 'For' cyclic sentences twice to place nibbles into matrix. Code is shown in Figure 3.1.

```
51  unsigned int bit_16_to_4(unsigned int input_16bit,unsigned int output_matrix[2][2]){
52      int buffer2,buffer3;
53      // buffer2= input_16bit;
54      // buffer3= input_16bit >>8;
55      for(int i= 0; i<2; i++){
56          buffer2= input_16bit;
57          buffer2 >>= 4*i;
58          output_matrix[1-i][1] = buffer2 & 0x000F;
59      }
60
61      for(int j=0; j<2; j++){
62          // buffer3= input_16bit >> 8;
63          buffer3 = input_16bit >> 4*(j+2);
64          output_matrix[1-j][0] = buffer3 & 0x000F;
65      }
66
67      return output_matrix;
68  }
```

Figure 3.1 16-bits into 4 nibbles

Next, transform matrix of nibbles back to 16-bits output. It is easy to implement it through bitwise shift and OR operation. Code is shown in Figure 3.2.

```
150  unsigned int Nibbles_to_bit16(unsigned int Nibbles_Matrix[2][2], unsigned int bit16_out){
151      unsigned int buffer4, buffer5, buffer6, buffer7;
152      buffer4 = Nibbles_Matrix[0][0];
153      buffer5 = Nibbles_Matrix[1][0];
154      buffer6 = Nibbles_Matrix[0][1];
155      buffer7 = Nibbles_Matrix[1][1];
156
157      buffer4 <<= 4;
158      buffer6 <<= 4;
159      buffer4 = buffer4 | buffer5;
160      buffer6 = buffer6 | buffer7;
161
162      buffer4 <<= 8;
163      bit16_out = buffer4 | buffer6 ;
164
165      return bit16_out;
166  }
```

Figure 3.2 nibbles back to 16-bits output

(2) substitution

With twice 'For' cyclic sentences, it is convenient to substitute values in original array. The substitution-box could be set as a global array variables (Figure 3.3). The original nibbles refers to queue number in array, then the output would be designed substitution. Both codes of 'NibbleSub' operation and 'Inverse NibbleSub' have been placed in Figure 3.4 and Figure 3.5.

```
4  int NibbleSub[16]={0xE,0x4,0xD,0x1,0x2,0xF,0xB,0x8,0x3,0xA,0x6,0xC,0x5,0x9,0x0,0x7};
5  int NibbleSub_Inverse[16]={0xE,0x3,0x4,0x8,0x1,0xC,0xA,0xF,0x7,0xD,0x9,0x6,0xB,0x2,0x0,0x5};
```

Figure 3.3 global array variables for substitution-box

```
72  unsigned int SubNibble(unsigned int input_Nibble[2][2],unsigned int output_Nibble[2][2]){
73
74      // int NibbleSub[16]={0xE,0x4,0xD,0x1,0x2,0xF,0xB,0x8,0x3,0xA,0x6,0xC,0x5,0x9,0x0,0x7};
75      for(int i=0;i<2;i++)
76      {
77          for(int j=0; j<2;j++){
78              output_Nibble[i][j]= NibbleSub[input_Nibble[i][j]];
79          }
80      }
81      return output_Nibble ;
82  }
```

Figure 3.4 'NibbleSub' operation

```
84  unsigned int Inverse_Nibble(unsigned int input_Nibble[2][2],unsigned int output_Nibble[2][2]){
85
86      // int NibbleSub_Inverse[16]={0xE,0x3,0x4,0x8,0x1,0xC,0xA,0xF,0x7,0xD,0x9,0x6,0xB,0x2,0x0,0x5};
87      for(int i=0;i<2;i++)
88      {
89          for(int j=0; j<2;j++){
90              output_Nibble[i][j]= NibbleSub_Inverse[input_Nibble[i][j]];
91          }
92      }
93      return output_Nibble ;
94  }
```

Figure 3.5 'Inverse NibbleSub' operation

(3) shift row

It is conenient to implement value transmittion in second row through assignment statement of array variables. Code is placed in Figure 3.6.

```
unsigned int Shiftrow(unsigned int input_matrix[2][2], unsigned int output_matrix[2][2]){

    output_matrix[0][0]=input_matrix[0][0];
    output_matrix[0][1]=input_matrix[0][1];
    output_matrix[1][0]=input_matrix[1][1];
    output_matrix[1][1]=input_matrix[1][0];
    return output_matrix;
}
```

Figure 3.6 shift row

(4) MixColumn – polynomial multiplication and addition in GF(2^4)

Addition is convenient to complete through XOR operation. Code is placed in Figure 3.7.Multiplication is consist of simply multiplication and modulo arithmetic. Code is placed in Figure 3.8.

```
106 unsigned int Mixcolumn(unsigned int input_matrix[2][2], unsigned int output_matrix[2][2]){
107     unsigned int constant_matrix[2][2] = {0b0011, 0b0010, 0b0010, 0b0011};
108     output_matrix[0][0]=pol_multi(input_matrix[0][0],constant_matrix[0][0]) ^ pol_multi(input_matrix[1][0],constant_matrix[0][1]);
109     output_matrix[1][0]=pol_multi(input_matrix[0][0],constant_matrix[1][0]) ^ pol_multi(input_matrix[1][0],constant_matrix[1][1]);
110     output_matrix[0][1]=pol_multi(input_matrix[0][1],constant_matrix[0][0]) ^ pol_multi(input_matrix[1][1],constant_matrix[0][1]);
111     output_matrix[1][1]=pol_multi(input_matrix[0][1],constant_matrix[1][0]) ^ pol_multi(input_matrix[1][1],constant_matrix[1][1]);
112     return output_matrix;
113 }
```

Figure 3.7 polynomial addition in GF(2^4)

```
21  unsigned int pol_multi(unsigned int input_pol1,unsigned int input_pol2) {
22
23      unsigned int result=0b0000;
24      unsigned int buffer1;
25      unsigned int divided=0b10011;
26      int n=4,m;
27      for(int i=0; i<n; i++) {
28          m= input_pol2 & 1;
29          input_pol2 >>= 1;
30          if(m){
31              buffer1 = input_pol1;
32              buffer1 <<= i;
33              result = result^buffer1;
34              }
35
36      }
37
38      // buffer2=Length(result);
39      // buffer3=Length(divided);
40      while(Length(result)>=Length(divided)){
41
42          result = result^(divided << (Length(result)-Length(divided)));
43      }
44
45
46      return result;
47
48
49  }
```

Figure 3.8 polynomial multiplication in GF(2^4)

By judging last bit of input_2, when the output is 'true', the input_1 would be shifted to correct position and oprate XOR with previous arithmetic result. Using 'For' cyclic sentence and bitwise shift operation to judge through every bits of input_2.

In programme of modulo arithmetic, the bit position difference refers to highest power element in polynomial. By bitwise shift with corresponding number and XOR operation, the modulo arithmetic is implemented. With conditional loop statement, the modulo arithmetic would keep operating until a irreducible result.

The correct bit position could be abstract by declaring a function to detect length of polynomial, as shown in Figure 3.9.

```
 9  int Length(unsigned int n){
10      int c = 0 ; // counter
11      while (n)
12      {
13          c++ ;
14          n >>= 1 ;
15      }
16      return c ;
17  }
```

Figure 3.9 detection for length of polynomial

(5)key-schedule and key-addition.
Key-schedule could be implemented directly through assignment statement of array variables and XOR operation.

Implementation of key-addition only needs XOR operation for addition and twice'For' cyclic sentences for operating through every elements in arrays, which is the same as before. Code is placed in Figure 3.10.

```
115 unsigned int KeyAddition(unsigned int input_block[2][2],unsigned int Ki[2][2], unsigned int output_block[2][2]){
116
117     for(int i=0; i<2; i++){
118         for(int j=0; j<2; j++){
119             output_block[i][j] = input_block[i][j] ^  Ki[i][j];
120         }
121     }
122     return output_block;
123 }
124
125 unsigned int KeySchedule(unsigned int K_0[2][2], unsigned int K_1[2][2], unsigned int K_2[2][2]){
126     unsigned int w0, w1, w2, w3, w4, w5, w6, w7, w8, w9, w10, w11;//round keys
127     w0 = K_0[0][0];
128     w1 = K_0[1][0];
129     w2 = K_0[0][1];
130     w3 = K_0[1][1];
131     w4 = w0 ^ NibbleSub[w3] ^ 0b0001;
132     w5 = w4 ^ w1;
133     w6 = w5 ^ w2;
134     w7 = w6 ^ w3;
135     w8 = w4 ^ NibbleSub[w7] ^ 0b0010;
136     w9 = w8 ^ w5;
137     w10 = w9 ^ w6;
138     w11 = w10 ^ w7;
139     K_1[0][0]=w4;
140     K_1[1][0]=w5;
141     K_1[0][1]=w6;
142     K_1[1][1]=w7;
143     K_2[0][0]=w8;
144     K_2[1][0]=w9;
145     K_2[0][1]=w10;
146     K_2[1][1]=w11;
147     return K_1, K_2;
148 }
```

Figure 3.10 key-schedule and key-addition

The input is '9c63' and key is 'c3f0'. The result is placed in Figure 3.11.

```
plaintext is 0x9c63
9 c 6 3
0 e 3 e
encoded text is 0x72c6
7 2 c 6
0 e 3 e
decoded text is 0x9c63
```

Figure 3.11 the result of Encryption and decryption

- Is your design vulnerable to timing analysis attacks? Explain your answer

No. Generally, the conclusion of correct key in timing analysis attacks requires linear relationship between practical execution time and predicted execution time for various inputs and keys.

In Mini-AES, Byte-Substitution provides the non-linearity component, while ShiftRow and MixColumn provide the linear component. Therefore, after multiple rounds, a certain result would has many possible combination with unperfect linear relation. Key-Addition mixes the round keys, which are produced from inputed key for different operation rounds, into encryption.

Timing analysis attacks could conclude a key. However, as there are still many possible combination of keys, in which only one of them is the correct one, the conclusion may be not the correct one. Therefore, it is not vulnerable to timing analysis attacks.

Reference

[1] Montgomery, D.C. and Runger, G.C. (2013) “Large-Sample Confidence Interval for a Population Proportion,” in Applied Statistics and probability for engineers. 6th edn. John Wiley & Sons Ltd, England, 291-294.