A Primer on Timing Side-Channel Attack

Leraning outcomes

At the end of this tutorial, your should be able to:

1. Describe the main principles of timing analysis attacks on embedded devices.
2. Explain how the Law of big numbers can be used to remove the effect of uncontrolled conditions or system parameters on timing analysis.
3. Describe how statistical metrics such as Pearson Correlation Coefficient are employed in the context of timing analysis.

1. Introduction:

Timing attacks (TA) are one of the most widely used side-channel analysis (SCA) method, it allows the extraction of the secret key of a cryptographic algorithm, for example, by exploiting the dependency of the algorithm's execution time depends on the binary value. Although, timing attacks pose a clear security threat and have been successfully applied at both the hardware and software levels of embedded systems, they are frequently overlooked the design phase because of their intrinsic dependency on the implementation and can inadvertently by compiler optimizations.

The generic procedure of timing attacks consists of three phases:

1) *Model Building*: Build a model of the system which emulate its relationship between the secret key and its timing characteristics (e.g. the execution time of its software algorithm)
2) *Experimental Data Gathering*: Perform experiments to measure the timing metrics (e.g. the execution time of the algorithm) of the system
3) *Statistical Analysis*: Compare the experimental data (e.g. the actual execution time) with the predicted value obtained from the model (e.g. the expected execution time) in order to find the key

In this exercise, we will give a number of examples to illustrate the principles of timing attacks

2. Basic principles of timing attacks

(Read this section carefully then answer the questions at the end)

This example demostartes how to carry a timing attack (as described above), it considers a simple case of an embdded device excuting (Algorithm 1) shown in figure 1, which has a secert key(KEY),

```
MES = IN XOR KEY;      // Exclusive-OR the input with the secret Key
FOR EACH b BIT in MES // For each bit of the resulting value (MES):
{
 IF (b == 1)
   routine();          // Perform a certain calculation only if bit = 1
}
```

Figure 1: Algorithm 1

The device in question is shown in figure 2, it has a 4-bit input, and store a 4-bit secret key

Input
Key
Out

Figure 2: An embedded device running Algorithm 1

2.1. Building a timing model

Algorithm 1 performs a bit-wise Exclusive-OR between a user-input (IN) and the secret key (KEY) and stores it in the variable MES. Each bit of MES is checked; if the bit is 1 a calculation (routine) is performed, if the bit is 0 the calculation is not performed. In this example, the execution of routine () takes 1ms, in this case, a simple model can be constructed as follows:

$$expTime = 1 \times HW(MES)\ ms \qquad (1)$$

Where

$expTime$: is the expected execution time

$HW(MES)$: is the Hamming weight of the Exclusive-OR the input with the secret Key

Table 1 illustrate how the expected execution times computed using the model given in equation (1), assuming the both input and the key are 4-bit long

Table 1. Execution Time Prediction for different Key guesses

User input IN	**(Actual) Exec. time**	**Time Prediction for KEY=0000**	**Time Prediction for KEY=0001**	**…**	**Time Prediction for KEY=1101**	**Time Prediction for KEY=1110**	**Time Prediction for KEY=1111**
0001	2 ms	1 ms MES = 0001	0 ms MES = 0000	…	2 ms MES = 1100	4 ms MES = 1111	3 ms MES = 1110
0010	4 ms	1 ms MES = 0010	2 ms MES = 0011	…	4 ms MES = 1111	2 ms MES = 1100	3 ms MES = 1101
0011	3 ms	2 ms MES = 0011	1 ms MES = 0010	…	3 ms MES = 1110	3 ms MES = 1101	2 ms MES = 1100
0100	2 ms	1 ms MES = 0100	2 ms MES = 0101	…	2 ms MES = 1001	2 ms MES = 1010	3 ms MES = 1011

2.2. Experimental Data Gathering

In order to measure the execution time of the algorithm, we will use the code provided in figure 2, which emulates the behaviour of Algorithm 1, for a 4-bit key

```
-----------------------------------------------------------------------------
%the KEY
key = 13; %Key=1101


%user inputs
IN = [1 2 3 4];
%number of user inputs
numIN = length(IN);

excTime = zeros (numIN,1);
predTime = zeros(numIN,16);
%precomputed Number of ones for all 0:15 values
HWTab = sum(dec2bin(0:15).' == '1');

for iCtrlIN = 1:numIN
    %Execution times (number of ones of the result of the XOR between the key
    and the user inputs):
     excTime(iCtrlIN)  = HWTab(bitxor(uint8(IN(iCtrlIN)),uint8(key))+1);
    for iCtrlKEY = 0:15
      %predicted execution times (number of ones of the result of the XOR
      between all the possible keys and the user inputs):
         predTime(iCtrlIN,iCtrlKEY+1)  =
          HWTab(bitxor(uint8(IN(iCtrlIN)),uint8(iCtrlKEY))+1);
    end
end
%display execution times
excTime
%display predicted execution times
predTime
-----------------------------------------------------------------------------
```

Figure 2: MATLAB Emulation of Algorithm 1

The code in figure 2 will produce two outputs:

1) The expected execution time (*excTime*) according to the model in equation (1) computed based on the secret key
2) The actual execution times (*predTime*) for all possible values of the key

2.3. Data Analysis

To guess the key, the attacker measures the execution time for a given input IN. Since the attacker knows the algorithm, he also knows that the execution time depends on individual bits of MES; hence, he can predict how much time the algorithm would take for an arbitrary key KEY and input IN. Trying all possible KEY values he can find the one that corresponds to the real measurements.

Task 1:

1. Read the code carefully, and then execute it. Compare the obtained output with that given in Table.1
2. Change the input vector to IN = [1] and execute the code again. Explain any differences between your results and the one shown above
3. Will you be able to guess the correct key if you use the input vector in step 2? Explain your answer.

3. The law of large numbers

(Read this section carefully then answer the questions at the end)

Let us consider the same example above with one modification, in this case, the attacker can only control a subset of the device’s inputs as shown in figure 3. In such a scenario, he/she would not be able to measure directly the execution time needed to process the input bits he/she controls, instead, the attacker can only measure the execution time of the whole algorithm, which is affected by the values of the remaining uncontrolled inputs

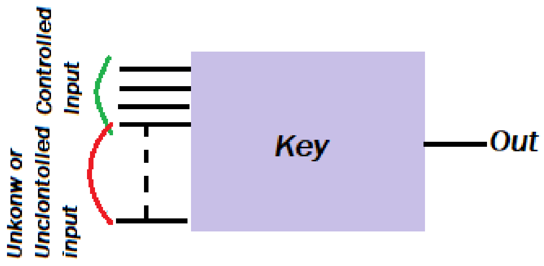

Figure 3: An embedded device

Waging a timing attack in this case is more difficult, but still feasible. To explain this, the attacker can exploit the law of large numbers "*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*". This simply means that in order to capture the dependency between the value of the secret key and the small number of inputs, controlled by the attacker, he/she should fix the value of their input bit, executes the algorithm a large number of times, and then computes the average of the execution times. The latter will tend to a constant value, that is only dependent on the attacker's controlled bits.

Let us assume for example that the input and the key are 16 bits each, and the attacker can only control 4-input bits, when they set to $IN_{3:0}$ = 0010. Let us also consider that the first nibble of the key is $KEY_{3:0}$ = 1010.

In this case, the expected execution time of *Algorithm 1* is 1 + 6 = 7 ms, where 1 is the execution time of the $MES_{3:0}$ ($MES_{3:0}$=1000) and 6 the average execution time of the remaining 12 bits (i.e. (the uncontrolled inputs) due to the law of large numbers.

In this case we can deduce the following general equation for *Algorithm 1*

$$expTime = HW(MES_{0:E})ms + 0.5\ (S - E) \qquad (2)$$

Where:

E is the number of bits controlled by the adversary,

S: The total number of input bits

Table 2 shows the execution time for all the combinations of $IN_{3:0}$ assuming $KEY_{3:0}$ = 1010, based on equation 2, where S=16, E=4.

Table 2. Execution Time Prediction for different input value of the first nibble.

$Key_{3:0}$	Input $IN_{3:0}$	$MES_{3:0}$	Number of ones in $MES_{3:0}$	Expected Execution Time [ms]
1010	0000	1010	2	8
1010	0001	1011	3	9
1010	0010	1000	1	7
1010	0011	1001	2	8
1010	0100	1110	3	9
1010	0101	1111	4	10
1010	0110	1100	2	8
1010	0111	1101	3	9
1010	1000	0010	1	7
1010	1001	0011	2	8
1010	1010	0000	0	6
1010	1011	0001	1	7
1010	1100	0110	2	8
1010	1101	0111	3	9
1010	1110	0100	1	7
1010	1111	0101	2	8

To demonstrate this principle, we will use the algorithm shown in figure 4.

```
-----------------------------------------------------------------------
%define set of colours for visualization
colors = ['r','b','k','g'];

%precomputed hamming weights (number of ones) for all 0:65535 values
HWTab = sum(dec2bin(0:65535).' == '1');

%The Key
key = hex2dec('6CEA');

%Number of measurements for each combination of IN
N = 3000;
acumExcTime = zeros(16,N);

figure;
hold on;

%iterate over all the combinations of the first nibble of input IN
for iCtrlIN = 0:15
 %iterate over N measurements
 for iCnt = 1:(N-1)

 %generate random input IN of 16 bits:
 randomInput = round(rand*(2^16-1));
 %create a mask to remove the first nibble from the random input
 mask = bitcmp(15,'uint16');
 %apply the mask to random input:
 maskedInput = bitand(randomInput,mask);
 %replace empty nibble with the controlled part of the input(iCtrlIN):
 input = bitor(uint16(maskedInput),uint16(iCtrlIN));
 %Simulate execution of the algorithm
 MES = bitxor(uint16(input),uint16(key));
 %obtain execution time (number of bits of MES)
 excTime = HWTab(MES + 1);

 %Accumulate the execution time of all the N times that the algorithm
 %is executed for the same value of IN):
 acumExcTime(iCtrlIN+1,iCnt+1) = acumExcTime(iCtrlIN+1,iCnt)+ excTime;
 end
 %Plot the progress of the average execution time at each execution of
 %the algorithm for each combination of IN using different colours:
 plot(1:N,acumExcTime(iCtrlIN+1,:) ./ (1:N),colors(mod(iCtrlIN,4)+1));
end
hold off;
xlabel('N of executions','FontSize',14);
ylabel('Average number of bits in MES (Average time)','FontSize',14);
set(gca,'FontSize',14);

%Display the average execution time obtained after the N measurements:
avgExcTime = acumExcTime(1:16,N) ./ N
```

Figure 4: MATLAB Scripts to Emulate the Effect of the law of large numbers

Task 2:

1. The code in figure 4 computes the average execution time when for a specific key, read it carefully and execute, then compare your results with that listed in Table 2
2. Using equation 2, compute the execution time if the first nibble of the key is set to “1111”, and then change this value in the code and verify that the average execution time obtained in both cases are approximately the same
3. 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.
4. Explain the significance of the minimum value of N, in the context of a practical side channel attack.

4. Statsitical Anslysis

The example shown in section 2 relies on finding the key through manual comparsion between the actual excution time and that computeted by the model (in equation 1), such a manual approach may not be feasble in practice (e.g. when the key space is large). Therfore, side-channel attacks typically use statistical metrics to establish if dependency exists between two variables. In this example, we will use the Pearson Correlation Coefficient (PCC), which shows if there is any linear dependency between two variables, i.e. if two variables x and y have a relationship y = ax + b (a,b are constants), then PCC will give a result equal to 1, otherwise PCC will give a much smaller value.

To demonstrate how the PCC metric can be used to automate timing analysis, we will use the code in figure 5. The latter is able to find the first nibble of 16-bit length key of the device shown in section 3, which is running *Algorithm 1*.

```
----------------------------------------------------------------------
%define set of colours for visualization
colors = ['r','b','k','g'];

%precomputed hamming weights (number of ones) for all 0:65535 values
HWTab = sum(dec2bin(0:65535).' == '1');

%The Key
key = hex2dec('6CE1');

%Number of measurements for each combination of IN
N = 10;
acumExcTime = zeros(16,N);

figure;
hold on;

%iterate over all the combinations of the first nibble of input IN
for iCtrlIN = 0:15
 %iterate over N measurements
 for iCnt = 1:(N-1)

 %generate random input IN of 16 bits:
 randomInput = round(rand*(2^16-1));
 %create a mask to remove the first nibble from the random input
 mask = bitcmp(15,'uint16');
 %apply the mask to random input:
 maskedInput = bitand(randomInput,mask);
 %replace empty nibble with the controlled part of the input(iCtrlIN):
 input = bitor(uint16(maskedInput),uint16(iCtrlIN));
 %Simulate execution of the algorithm
 MES = bitxor(uint16(input),uint16(key));
 %obtain execution time (number of bits of MES)
 excTime = HWTab(MES + 1);

 %Accumulate the execution time of all the N times that the algorithm
 %is executed for the same value of IN):
 acumExcTime(iCtrlIN+1,iCnt+1) = acumExcTime(iCtrlIN+1,iCnt)+ excTime;
 end
 %Plot the progress of the average execution time at each execution of
 %the algorithm for each combination of IN using different colours:
 plot(1:N,acumExcTime(iCtrlIN+1,:) ./ (1:N),colors(mod(iCtrlIN,4)+1));
end
hold off;
xlabel('N of executions','FontSize',14);
ylabel('Average number of bits in MES (Average time)','FontSize',14);
set(gca,'FontSize',14);

%Display the average execution time obtained after the N measurements:
avgExcTime = acumExcTime(1:16,N) ./ N
%Matrix to store the expected execution times for the combinations of all the
possible values of IN and keys:
timeModel = zeros(16,17);
%the first column corresponds to the average execution times obtained
%after the N measurements:
timeModel(:,1) = avgExcTime;
for iCtrkey = 0:15
    for iCtrlIN = 0:15
      %The expected execution time is the expected number of ones in the result
      of the XOR between the input nibble and the guessed key, plus the
      expected number of ones in the random part (12/2)
         timeModel(iCtrkey+1,iCtrlIN+2) =
HWTab(bitxor(uint8(iCtrlIN),uint8(iCtrkey)) + 1) + 6;
    end
end
timeModel

%Matrix of Pearson correlation coefficients:
Rm = corrcoef(timeModel);
%First row of Rm contains the correlation coefficients between the values of
avgExcTime and the expected execution times for all the possible values of
KEY3:0
Rc = Rm(1,2:17);
%The entry of Rc with the highest positive value corresponds to the guessed
key (the first entry of Rc is 1 and corresponds to the autocorrelation of
avgExcTime, therefore is discarded)
[corr,idx] = max(Rc);
guessedKeyNibble = idx-1
```

Figure 5: MATLAB Script for a timing attack employing the Pearson Correlation Coefficient

Task 3:

1. *Execute the code in figure 5, for different values of the key, verify the code is still able to guess correctly the first nibble of the key (the first nibble is the right most digit (i.e. 6))*
2. This example uses Pearson Correlation Coefficient, what other statistical metrics can be used in this case. Justify your answer with experimental evidence.