Assignment 2: Side-Channel Analysis and Fault Injection

Amar Lakshya (AXX945), Calvin Menezes (CXM1010)

School of Computer Science, University of Birmingham

---

1. Task1

The average number of cycles per execution of the exponentiation for e = 17 is: **9,224,516** cycles.
Number of cycles (C) consumed by exponentiation with a 1024-bit exponent assuming 50% of the bits to be 1 was estimated using the following method:

- A square (S) operation would be performed for every bit that is being processed and an additional multiply operation (SM) for bits that are 1.
- Hence, at every iteration of the exponent bit, an S or SM operation takes place.
- We observed the number of cycles needed for one S and one SM operation. $C_S = 1840048$ and $C_{SM} = 3680264$ were obtained by setting $e = 0x2$ i.e. 10_2 and $e = 0x3$ i.e. 11_2 respectively. $C_{MSB} = 4984$ (found by setting $e = 0x1$) was subtracted from both C_S and C_{SM} because the MSB is processed only once and shall be added to the cycle count calculation explicitly.
- For estimating C_{1024} with a 1024-bit e, since the most significant bit is 1, the rest of the bits must be 01010101 and so on to achieve a probability of 50% on bit values.
- Therefore, number of cycles is:

$$C_{1024} = C_{MSB} + C_{S2} + C_{SM3} + C_{S4} + C_{SM5} + \cdots + C_{S1024} \quad (1)$$
$$C_{1024} = C_{MSB} + (512 * C_S) + (511 * C_{SM}) \quad (2)$$
$$C_{1024} = 4984 + (512 * 1840048) + (511 * 3680264) \quad (3)$$
$$C_{1024} = 2822724464 \quad (4)$$

Therefore, the extrapolated cycles for a 1024-bit exponent exponentiation where 50% of bits are 1 is: **2,822,724,464** cycles.
The solution for the task can be found in the *crypto.c*.

2. Task2

2.1. Oscilloscope Capture

The image of the RSA operations can be seen below:

Figure 1: The picture of oscilloscope showing the first 6 operations of RSA

2.2. Manual Extraction

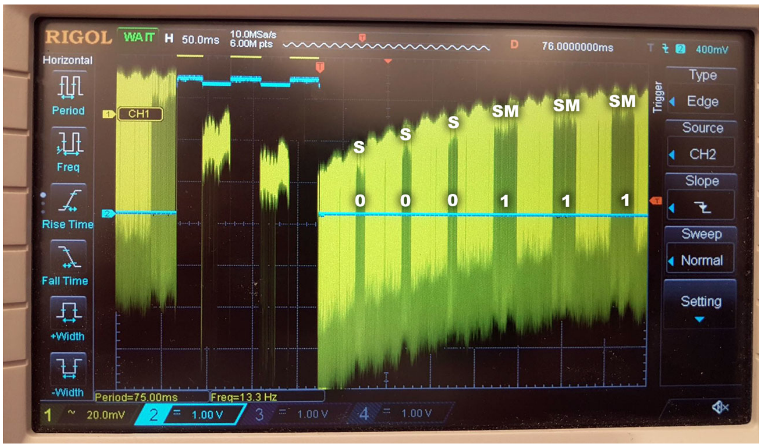

Figure 2: Superimposed picture showing operations as seen in the trace

- Based on the square-and-multiply (SaM) algorithm, the for-loop contains a square (S) operation that takes place in every iteration and a multiply (M) operation that is done only when the current exponent bit is 1.
- The initial bit of the exponent is always 1 and the loop starts from the 2nd bit onwards (Left to right).
- The screenshot above from an oscilloscope shows variation in power consumption as the algorithm runs.
- The thinner lines plotted indicate a square operation and the thicker lines indicate squaring followed with multiplication.
- The sequence of operations from the graph are S, S, S, SM, SM, SM that translates to the bits being 0, 0, 0, 1, 1, 1.

- Therefore, the upper six bits of the exponent e are 100011.

2.3. Acquired Trace

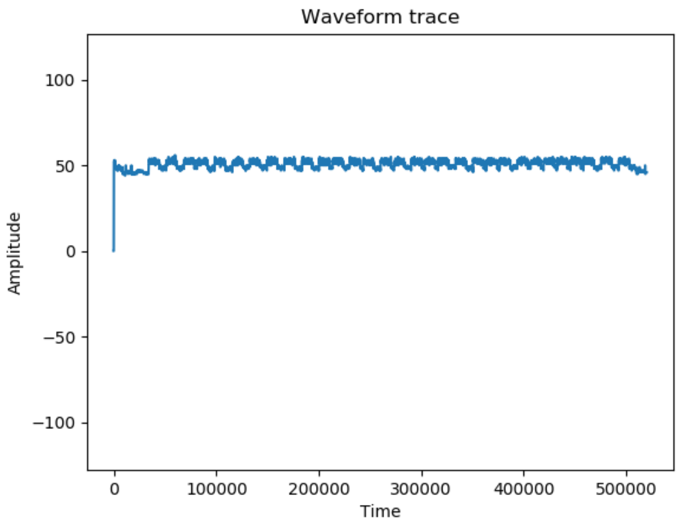

Figure 3: Graph of the acquire trace in full scale

The graph of the acquired trace can be found above and the program file *task2.3.py* contains the program.

2.4. SPA

The following is the output of the submitted *task2.4.py* program:

```
Iterating through 520000 sample points from trace_filtered.dat
Analyzing waveform... ... ...

----- Exponent extracted -----

        Binary: 1110 1111 1110 1010 1001 1001 0000 0111
                Hex: efea9907
```

3. Task3

3.1. Overlapping Traces

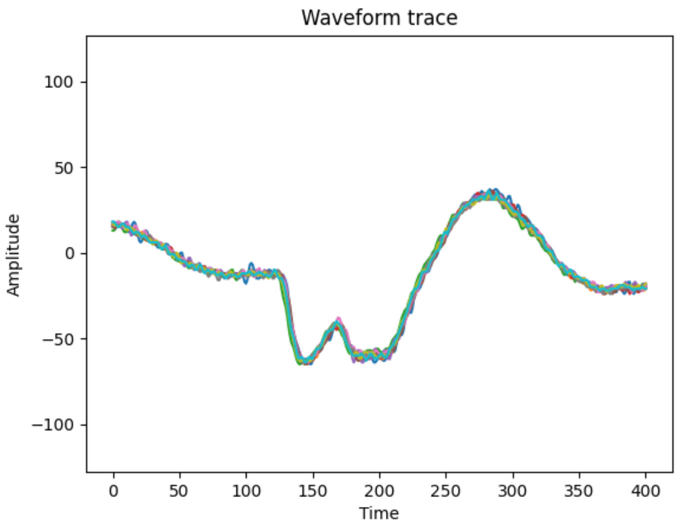

Figure 4: Graph of superimposed 10 traces

The plot of the first 10 traces can be seen above.

The program file *task3.py* contains the program for the task. The corresponding output can be seen below:

```
byte value for trace 0: D8
byte value for trace 1: E8
byte value for trace 2: 42
byte value for trace 3: 3D
byte value for trace 4: 96
byte value for trace 5: E5
byte value for trace 6: CB
byte value for trace 7: 0B
byte value for trace 8: 1F
byte value for trace 9: 33
```

3.2. AES XOR

The following function implements the required functionality (the complete program can be found in the submitted zip):

```
def AES_XOR(x, k_prime):
    return SBOX[x ^ k_prime]
```

3.3. Significant Bit

The method to extract the *MSB* involves two steps where we first convert an *integer* to a *bit-field* and then we get the first element of the *bit-field*. The following is an implementation of the function in *python*:

```
def int_to_bits(n):
    return [n >> i & 1 for i in range(BIT_SIZE-1,-1,-1)]

def MSB(n):
    return int_to_bits(n)[0]
```

3.4. DPA

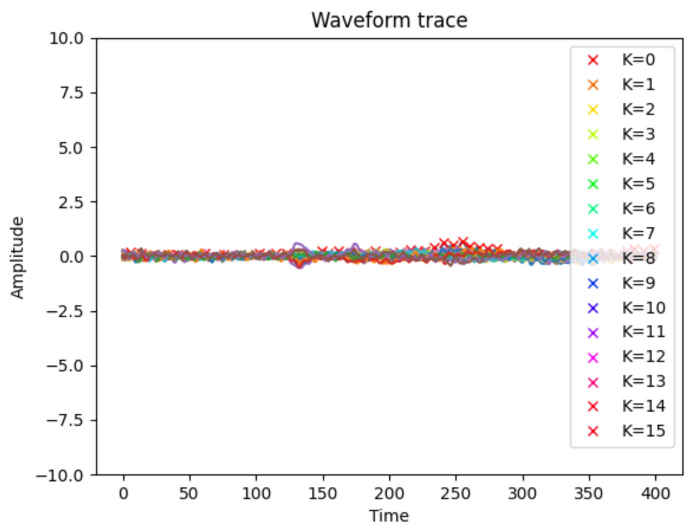

Figure 5: Graph of superimposed difference-of-means

The submitted program file *task3.4.py* contains the program to produce the overlapping graphs of all difference-of-means curves as seen above and the zoomed-in part showing the candidate key $K = 0$ and $max\_peak = 0.6620911131952134$ can be seen below.

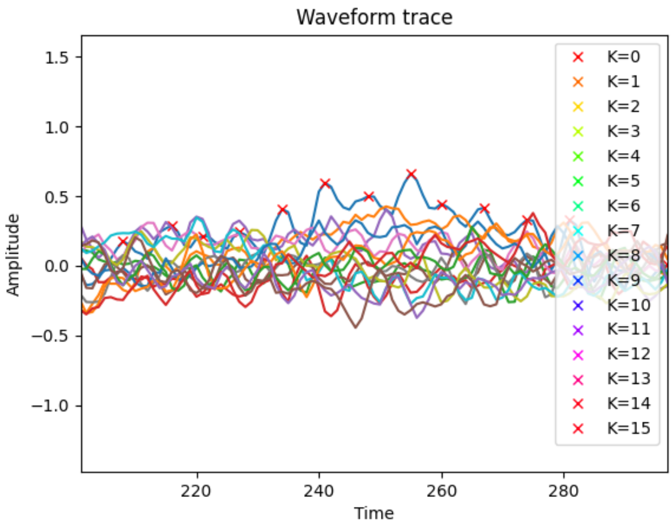

Figure 6: Graph of zoomed-in part of difference-of-means

4. Task4

4.1. Fault Injection on RSA-CRT

From the given values $n = 91, e = 5, S = 48, \overline{S} = 35, x = 55$ we calculate the valid signature y by:

$$y = x^e \mod n \tag{5}$$
$$y = 55^5 \mod 91 \tag{6}$$
$$y = 48 \tag{7}$$

4.1.1. Using Bellcore Method

The bellcore method gives factor q by:

$$q = \gcd(S - \overline{S}, n) \tag{8}$$
$$q = \gcd(48 - 35, 91) \tag{9}$$
$$q = \gcd(13, 91) \tag{10}$$
$$q = 13 \tag{11}$$

we know $n = p * q$ therefore factor p is:

$$p = n/q \tag{12}$$
$$p = 91/13 \tag{13}$$
$$p = 7 \tag{14}$$

4.1.2. Using Lenstra Method

The Lenstra method gives factor q by:

$$q = \gcd(\overline{y}^e - x, n) \tag{15}$$
$$\tag{16}$$

From eq 1, we know that $y = 48$ and since $\overline{y} = y, \overline{y} = 48$, therefore

$$q = \gcd(48^5 - 55, 91) \tag{17}$$
$$q = \gcd(48 - 55, 91) \tag{18}$$
$$q = \gcd(7, 91) \tag{19}$$
$$q = 7 \tag{20}$$

we know $n = p * q$ therefore factor p is:

$$p = n/q \tag{21}$$
$$p = 91/7 \tag{22}$$
$$p = 13 \tag{23}$$

The major advantage of Lenstra method is that only the faulty value $\overline{S}$ needs to be known.

4.2. Counter Measures

4.2.1. Difference between Detection based and Algorithmic

- Detection-based countermeasures are based in detecting faults through hardware or software properties of faults while Algorithmic take help of specific algorithmic patterns and monitor changes in those patterns caused by faults.
- Since Algorithmic countermeasures can use techniques like time randomization, they are much harder to inject faults against.

4.2.2. RSA-CRT Countermeasures

- Detection-based countermeasures
  - A hardware-level approach could be to monitor power supply voltage and ensure it does not drop below a specified limit. If it does, the RSA computation can be halted. This is expensive since it requires monitoring of voltage at an extremely small time frame i.e. at nanosecond scale.
- Algorithmic countermeasures
  - As a countermeasure, the correctness of the signature can be verified by comparison of inverse result with the input. If the values turn out to be equal, no FI attack was performed.

4.2.3. SPA Prevention

There a number of countermeasures that can be taken to prevent the SPA attack:

- The Square and Always Multiply Algorithm can be used to conditionally throw away the result so that there's less contrast in the acquired power trace.
- Noise Addition or Signal Reduction can be implemented on the signal to reduce the SNR
- Balancing power consumption requires a lot of changes in hardware that create redundant circuits and is therefore not a preferred countermeasure.

4.2.4. Time Randomization

- In Both Fault Injections models namely *Flip-Bit* and *Stuck-at*, the faults are injected in order to alter the normal control flow of a program and since the instructions in a control flow are dependent on time, Time Randomization makes these injections to work.
- In Side-Channel attacks focus on recovering secrets through analysis of the Runtime. Since the Runtime of a program is based on time, Time randomization makes these attacks also harder to work.

4.2.5. DPA Effects

- Increasing the amplitude noise decreases the SNR and therefore each power trace will have lesser amount of signal and a greater number of traces will be required.
- If the attacker can acquire infinitely many traces then increasing noise below a certain point (after which the signal is no longer traceable) will not be a suitable countermeasure. However under a finite number of traces, this technique can function as an effective countermeasure.