Assignment 2: Side-Channel Analysis and Fault Injection

Hardware and Embedded Systems Security

January 5, 2018

Introduction

The goal of this assignment is to first implement a toy version of RSA on the MSP430, and then attack cryptographic algorithms running on the Microcontroller (µC) using Side-Channel Analysis (SCA). For the part of the assignment done in the lab (task 2), you will be supplied with a prepared µC board. We will also have a lab schedule on Canvas to organise the timeslots in the lab.

Your main deliverable for this assignment is a report (see below for exact tasks, length approx. 3–4 pages) plus the program code you developed. For solving this exercise, the following programming languages can be used: `C` and `C++` (no non-standard libraries please), `Java` (no non-standard libraries please), or `Python` (including `numpy` and `pyplot`, highly recommended). In any case, your code must be compilable and testable with the lecture VM. Precompiled files are not accepted. For plotting, I recommend `gnuplot` or `pyplot`, which can be installed using:

```
sudo apt-get install python-matplotlib
```

You will receive a `Python` template for communicating with the oscilloscope and Device Under Test (DUT). Please submit your solution via Canvas in a `.zip` archive named

```
groupXX-assignment2.zip
```

This archive should include the following files and folders:

report.pdf Your report / solutions for the non-programming questions (see below)

longnum_sam/crypto.{c,h} Program code for Task 1

rsa_spa Program code for Task 2

aes_dpa Program code for Task 3

This assignment contributes 25% to your final grade.

The grade for each task is determined by: correctness of results, quality of report, quality of code, creativity and excellence. The pen-and-paper problems are graded for correctness (results are correct and legible) and completeness (i.e. all intermediate values are given as required, explanations are clear, etc).

1 Implementing Square-and-Multiply (SAM) (15%)

Implement the Square-and-Multiply algorithm to compute $s^e \mod n$ of two long numbers in `C` for the MSP430. Use the supplied template and the functions from the longnum lib (`ln_add()`, `ln_multiply()`, `ln_square()`). Note that the template performs RSA signature verification for the short exponent $e = 17 = (10001)_2$. Hence, we here assume that e is passed as a 16-bit word, in reality e would be a long number as well. Note that your implementation should be correct for all other valid 16-bit exponents as well.

Note: The template already contains an implementation of modular reduction using Barrett's method, along with instructions on the usage. Do not re-implement modular reduction!

Report: Give the average number of cycles per execution of the exponentiation for $e = 17$. Extrapolate how many cycles a full 1024-bit exponent exponentiation would consume (assume that 50% of the exponent bits are 1).

2 Simple Power Analysis (SPA) on RSA (30%)

In this task, we target a are given an (toy) implementation of RSA (with only 256 bit operands and 32-bit key) for the MSP430, similar to the one you developed in Task 1. In this particular implementation, the developer has implemented an "optimization" for the squaring, which has made the squaring four times slower than a multiplication. Further note that for the modular reduction, Barrett reduction is used, which consumes an additional multiplications per modular reduction.

To trigger an RSA operation repeatedly, use the script `repeat_rsa.py`. Make sure to set your group number in this file first (`GROUP_NUMBER` should be set to your group number minus 1). `repeat_rsa.py` has to run as root, i.e., call it with `sudo python repeat_rsa.py`. Your tasks (first three subtasks have to be done in the lab) are:

1. Set up the oscilloscope so that it captures a full RSA operation. Channel 1 should be the side-channel trace, channel 2 the trigger signal (`P1.3`). For this task, `repeat_rsa.py` is useful. Take a screenshot (green "Printer" button) of the configuration (you need to connect a USB stick as storage to the front port of the oscilloscope). Also take a photograph of the measurement setup for your report.

2. For the first 5 bit of the exponent, manually extract the sequence of operations (Square = S, Multiply = M). With this, determine the value of the upper 6 bits of the exponent. This task is best solved using the oscilloscope user interface (stop, then use horizontal scale and position buttons). Also take a screenshot of the (zoomed) parts that belong to a multiplication and a squaring, respectively.

3. Plot a complete acquired trace, which is already provided in `trace_filtered.dat`. In the file `trace_filtered.dat`, the waveform is stored as a simple sequence of 8-bit signed integer values (-128–127). The provided script `load_trace.py` gives an example how to load such a file.

4. Write a program in the language of your choice (see above for recommendations) that automatically performs an SPA and extracts and prints the exponent (in binary and hex) for the file `trace_filtered.dat`. Since some filtering is required, you are this time given a pre-recorded and pre-processed trace `trace_filtered.dat`. The script `filter_trace.py` has been used to generate the filtered trace (this program is only given if you are interested, not required for solving the assignment).

3 Differential Power Analysis (DPA) on AES (35%)

In this task, your goal is to implement DPA on an AES reference implementation. To guarantee that all groups work on the same data, a set `aes_traces.zip` of 1300 traces is provided with the assignment. The file format is the same as for the previous task. The traces are already cut so that they only contain the clock cycle of interest. Each trace is 401 sample points long.

The traces are numbered starting at zero, i.e., `trace0.dat` is the first trace, and so on. In `inputs.txt`, the corresponding input values are given as hex ASCII values. Each line is the input for one trace, i.e., the first line is the input for `trace0.dat`, and so on. Your tasks are:

1. Implement a program that reads the first l traces, plots them (all overlayed into one plot), and loads prints the first byte of the corresponding input values. Include the result (the plot and the byte values) for $l = 10$ in your report.
2. Implement a function that takes the first input byte x, a key candidate byte $\hat{k}$, and returns the value after the AES S-Box, i.e., $y = S\left(x \oplus \hat{k}\right)$.
3. Implement a function that returns the *most-significant* bit of a given byte.
4. Connect the above functions to implement the actual DPA to recover the first byte of the key $\hat{k}$: Your program should take the trace set, `inputs.txt`, and the number of traces l as input. It should output the difference-of-means curves for all key candidates $\hat{k} = 0 \ldots 15$ (for time reasons, it is not required to test all values $\hat{k} = 0 \ldots 255$). Find the difference-of-means curve with the highest peak to identify the correct key candidate.

4 Pen-and-Paper Problems (20%)

Please include the solution to the following problems in the report. Note that similar questions may be asked in the exam, hence, avoid using any tool apart from pen & paper and a normal calculator. You are allowed to solve this by hand and include a (good quality, readable) scan or photo in the report.

4.1 Fault Injection (FI) on CRT-RSA (10%)

The goal of this task is to apply the Bellcore and Lenstra attacks covered in the lecture. You are given the following RSA-CRT parameters:

$$n = 91,\ e = 5$$

You are also given a correct signature $s = 48$ on the plaintext $x = 55$, and a faulty one $s' = 35$ on the same plaintext. Using both the Bellcore and Lenstra method, recover the factors p and q of n. Show your intermediate results. What is the advantage of the Lenstra method?

4.2 Countermeasures (10%)

In this task, we will deal with countermeasures against FI and SCA-based attacks. Briefly answer the following questions (3 sentences max. each):

1. Explain the main difference between detection-based and algorithmic countermeasures against FI.

2. Assume a fault in an RSA-CRT implementation can be induced by a power glitch. Describe (a) a detection-based and (b) an algorithmic FI countermeasure.

3. Describe how the SPA attack from Task 2 could be prevented.

4. Why does randomization in time make both FI and side-channel attacks harder?

5. Which effect does increasing the amplitude noise have on the number of required traces for a DPA or Correlation Power Analysis (CPA)? Assuming perfect time synchronization, is increasing the noise a suitable countermeasure if the adversary can acquire many traces?