Week 1: Power SCA

Deadline: Fri, November 28nd @ 23:59

The first week will be all about extracting and analyzing power traces from the target during the encryption rounds. With the right technique, you can **recover the encryption keys** by just observing the power consumption.

Overview

For this week, you will need to complete the following tasks. You will upload your solution with the code for each task and a report (preferably a single PDF with one section for each task).

- Task 0: 0 points
- Task 1: 1 point
- Task 2: 1 point (+0.25 for bonus)
- Task 3: 1 point
- Task 4: 3 points
- Task 5: 4 points
- Task 6 (Bonus): 1 point

IMPORTANT:

- It is a good idea to setup a **private** git repository (one per group) for the whole duration of the project, and give access to the the TAs (Github Usernames: AlviseDeFaveri, dyonwg)
- You will also need to upload your solution (typically a tarball of your git repo with code + report) at the end of the week on Canvas
- **ALL scripts must finish in less than ~15 minutes on the hwsec-cw machine.**
- Please avoid using the Chipwhisperer's Jupyter Notebook directly on the server – it can hog the machine and hang the connection with the CW
- Alternative solutions are welcome but you first need to perform the intended SCA to get full score
- Do not update the CW firmware yourself – if you suspect you need that, ask the TAs 🙂

Step 0: Interface Setup

Let's start by verifying that you can communicate with the LameDongle™.

SSH into `hwsec-cw` with your group's username. You should see a `(cw)` banner at the beginning of the bash command line. The chipwhisperer toolchain has been downloaded and installed for you, and can be found under `~/chipwhisperer`. In case you need additional software, please contact the TAs.

As a first step, run the `test_cw.py` in your home folder to check that you can connect to the CW. This script will simply initialize both the scope and the target of the CW, and reset them both together, before opening an IPython shell (the target in our case represents the LameDongle™). You can reuse this for later scripts as well, so take a look at the code inside it.

To send and receive bytes to/from the target, the CW library uses simpleserial. Specifically, a command 'd' (for DES) is implemented to encrypt the provided **8 bytes** of input data (plaintext), and the ciphertext is returned on the 'r' command. For example:

- **target.simpleserial_write('d', b"HWSEC_25")** to send the plaintext
- **ct = target.simpleserial_read('r', 8)** to receive the encrypted plaintext

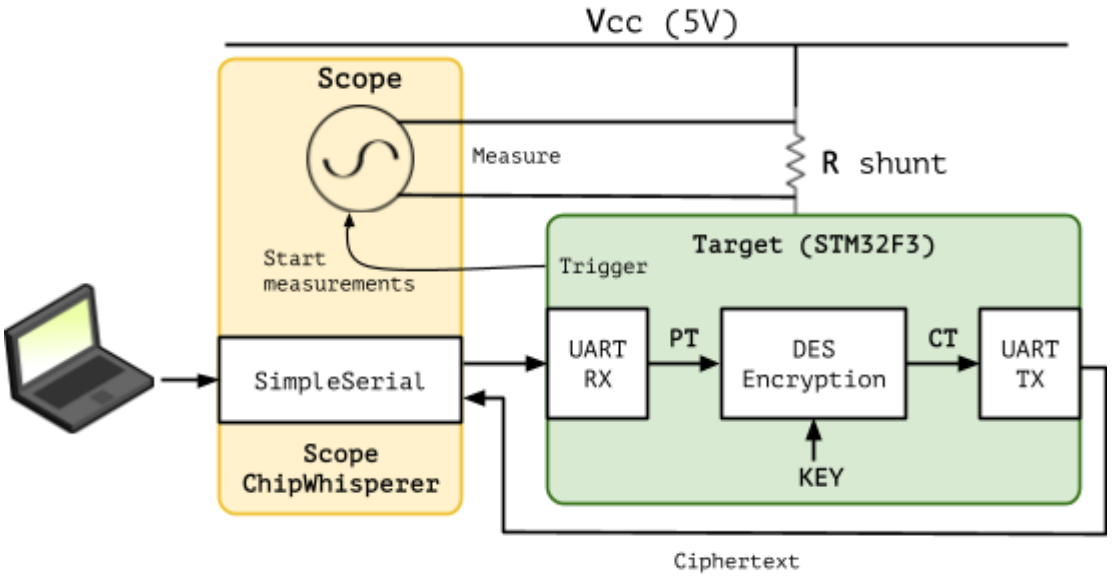
Task 0

Ensure that you can communicate correctly with the firmware by sending the following plaintexts and verifying that you receive the corresponding ciphertext.

Plaintext (hex)	**Ciphertext**
4142434445464748	ef 77 0c 97 ad 06 2c 75
4141414142424242	94 70 7c 83 8c aa df a7
4242424243434343	5f 1b 46 e2 85 2e ad f7

Step 1: Capturing power traces

Now that we know the target is responsive, let's try to sniff some power traces. Luckily, the ChipWhisperer (CW) provides everything out of the box as shown in the figure below. A shunt resistor is soldered on the target board to allow the CW's scope to make power consumption measurements. With the CW you can also send and receive messages over the UART interface of the STM32F303, which in our case represents the LameDongle™.



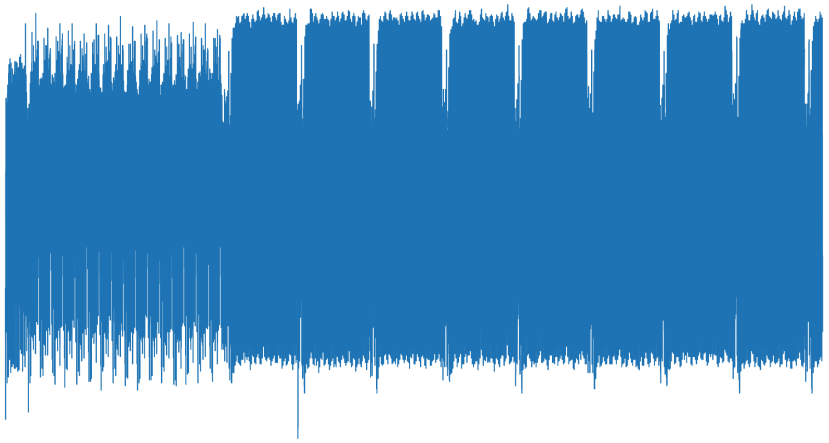
To perform a Power Side-Channel Analysis attack you need to measure the entire power consumption of the DES encryption. Fortunately, the scope included on the CW is able to store enough samples so that you can capture the entire DES encryption power consumption.

For simplicity, in the victim firmware we added a call to ***trigger_high()*** right after the plaintext/challenge is read from UART and a call to ***trigger_low()*** right after the DES encryption is completed. This was added to your benefit so you can always start capturing power traces in a well synchronized manner. Please notice that the CW scope will start capturing right after trigger_high() is called and it will stop after the configured amount of samples are captured. **trigger_low() will not stop your capture!** It is necessary only to prepare for the next trigger.

For capturing a trace have a look at the OpenADC API. The functions properties that you will need are:

- Offset
- Decimate
- Samples
- Arm
- Capture
- Get_last_trace

Try to obtain a trace similar to the following one. This was obtained by zooming out until all the trace is captured. Here you can have a helicopter view of all the encryption and the ciphertext print. This view is useful to zoom in only on the interesting part (i.e. encryption).



Task 1

Capture some power traces with different zoom/offsets and try to find where:

- The key scheduling is made
- The encryption is performed
- The ciphertext is printed to the UART

Deliverables

- Screenshots with some annotations about the 3 phases that you have to find.
- A small report in which you describe your intuitions.
- The python script used to capture the power trace

Hints

- UART is a very slow protocol
- DES encryption consists of 16 rounds
- DES key derivation consists of 16 rounds and it's always performed in this implementation (before the encryption)

Step 2: Define a Power Model

One critical component is missing before we can perform Power Side-Channel attacks: **a power model**. Most of the time, digital systems are implemented using CMOS logic gates. For example, a CMOS NOT gate is shown in the figure below. When V_{in} switches from 5V to 0V (image on the right) both transistors will change their state. For example, the top one switches from open circuit to short circuit (OFF->ON), thus bringing Vout to 5V. Bringing Vout from 0V to 5V requires energy since the parasitic capacitance must be charged.

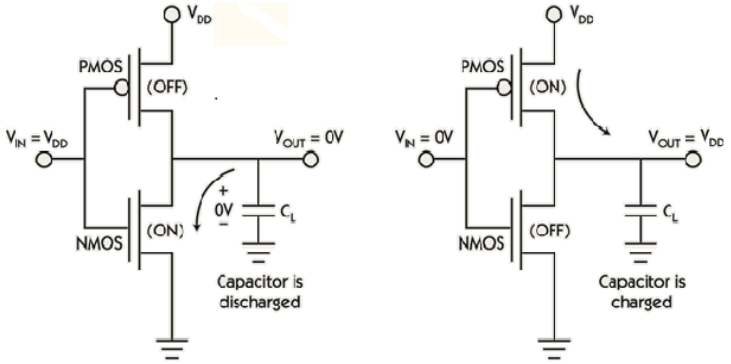


Fig. 2. CMOS inverter switching power analysis.

(Image source: EDN Inside DSP on Low Power: Designing Low-Power Signal Processing Systems)

If the above paragraph sounds too complex to you, don't worry! The important concept to remember here is that **a logic gate consumes power mainly when switching state**.

In other words, the instantaneous power consumption in a digital system is proportional to the number of transistors switching during a clock event.

More details can be found here:

- Introduction to Side-Channel Analysis [strongly recommended]
- CMOS Dynamic dissipation
- Paul Kocher et al. - Differential Power Analysis

After the video "Introduction to Side-Channel Analysis", you should be convinced that we can use a simpler power mode: the power consumption of a bus is proportional to the number of bits set to 1.

In other words, the power consumption is proportional to the hamming weight of the data passing in the bus.

Task 2

Now that you have a power model you can try to make a very simple analysis. Your goal is to find approximately where the plaintext is handled on the target. Remember that power consumption is proportional to the hamming weight!

Capture a set A of 100 power traces when the plaintext has the upper 32 bits set to 0 (0x0000XXXX where X means random – has to be different in each trace) **and a set B** of 100 power traces when the plaintext has the upper 32 bits set to 1 (0xFFFFXXXX). It's important to capture the traces of the entire encryption.

Average the two sets of traces to obtain an average trace for set A (tA_{avg}) and an average trace for set B (tB_{avg}). Then **plot** in the same figure one of the 200 power traces together with **abs(tA_{avg}-tB_{avg})**. You should see some spikes, and the biggest spike should be close to the beginning. Can you explain what this spike represents?

Deliverables:

- The aforementioned plot
- An explanation of why the spike is present, and why you think this can be useful
- The python script that you used to generate the plot

Bonus:

- Can you do the same for the **ciphertext**? If yes, attach a plot together with the script and a description of your analysis.

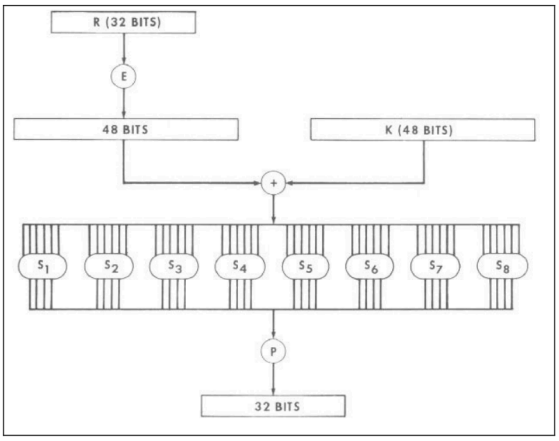
Step 3: Understanding DES

The last missing component is to understand where to attack DES. Have a look at the standard:

- https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf
- The DES Algorithm Illustrated

As you can see the key is expanded in 16 sub-keys, and K1 will be used only on the first round. We are interested in **leaking K1**, since if you know K1 you can easily recover the entire initial key. Can you explain why this is the case?

As shown in the figure below, during the first round, K1 is XORed with the expanded R. R is known since the plaintext is known.



(Image source)

If we suppose that the upper 6 bits of K1 are known, then **we can compute the output of S-Box 1**. This is a perfect target for our power analysis: by supposing a small portion of K1 (6 bits) we can compute the hypothetical output of an S-Box. In other words, by guessing 6 bits of the key, and by knowing the plaintext, we are able to compute 4 hypothetical bits of data that will be transferred over the bus. The idea is that by trying all the possible 6-bits combinations of upper K1, only one of them will match with all the power traces that we will capture.

You might ask why S-Boxes are a good target? Can one not simply attack the XOR operation? The XOR between R and K1 will be also sent on the bus, however, this operation is highly linear (small input difference = small output difference) compared to S-Boxes. Non-linearity is a desired feature, e.g. it helps to better distinguish between two values that are very similar to each other.

Task 3

Create the function sbox_out(sbox_num, plaintext, guess_k1)
Where:

- Sbox_num is a number from 1 to 8 used to indicate which S-Box output you want to compute
- Plaintext: the entire 64-bit input plaintext
- Guess_k1: the 6 bits of K1
- The return value is the 4-bit result of the selected S-Box

Deliverables:

- The script where you have implemented sbox_out
- Explain why finding K1 is enough to find the 64 bit DES key

Hints:

- You can re-use any implementation of DES that you find online. Just cite them.
- Make sure that your implementation is correct. This is a fundamental building block to do next tasks!
- To verify that it works, run the algorithm with the following parameters:

PT = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001
$\mathbf{K_1}$ = 000110 110000 001011 101111 111111 000111 000001 110010
$\boldsymbol{R_0}$ = 1111 0000 1010 1010 1111 0000 1010 1010
$\mathbf{E}(\boldsymbol{R_0})$ = 011110 100001 010101 010101 011110 100001 010101 010101
$\boldsymbol{K_1}+\mathbf{E}(\boldsymbol{R_0})$ = 011000 010001 011110 111010 100001 100110 010100 100111

You should get the following SBOX outputs:

$\boldsymbol{S_1(B_1)...S_8(B_8)}$ = 0101 1100 1000 0010 1011 0101 1001 0111

Step 4: DPA - Differential Power Analysis

Now you have all the tools to perform **Power Side-Channel analysis**!

Let's start from the DPA attack described in Paul Kocher et al. - Differential Power Analysis. DPA is a simple (and noisy) attack in which you basically:

1. Choose a set of values (random)
2. Collect the power traces of the encryption of each value
3. "Simulate" an encryption round of each value for each possible key
4. Check which of the "simulations" fits the observed data the most

To build such "simulations" you need to take each input and, for each possible key, calculate what would be the output of the first SBOX lookup. For each key, you will end up with two sets:

1. zero_inputs: Inputs for which the SBOX output's LSB = 0
2. one_inputs: Inputs for which the SBOX output's LSB = 1

You then collect the real power trace for each input. At this point, for each key, you have a list of traces coming from the zero_inputs (**zero_list** or 0-traces) and another list of traces associated with the one_inputs (**one_list** or 1-traces).

The analysis can then be implemented as follows:

```
For every S-Box:
    For every guess_key in {0..63}:
        one_list  = list of all the traces where the simulated sbox_out last bit is set to 1
        zero_list = list of all the traces where the simulated sbox_out last bit is set to 0
          // Take the max distance of the sample-by-sample avg for each list
          max_peak = max(abs(avg(one_list)-avg(zero_list)))
          If max_peak > best_beak:
                best_peak = max_peak
                Best_key  = guess_key
```

The idea is that, if the guess is correct, there is a specific moment during the trace in which all the 1-traces consume more on average than the 0-traces, because of the higher hamming weight, while everything else averages out.

Task 4

Perform DPA on your power traces and recover K1.

Deliverables:

- Your script to perform DPA to recover K1.
  IMPORTANT: Your script must do everything from the capture to printing the best K1 candidates! The only argument of your script is the number of traces to capture (so the script should be executed as ./dpa <n_traces>).

- A report describing your **results**, **how the attack works** and **how to run the script**. Notice that if we cannot run your script you will get 0 points for this part!

Hints:

- **DPA is quite noisy, don't worry if you don't always get the correct key**! Try to print the best 5 guesses for every sub key (6 bits) and ensure that the correct one is between them
    - Full score will be given only if, for all the K1 6-bit guesses, the correct one is among the 5 best candidates
- If you have doubts on the attack we suggest you read the original paper that proposed this attack
- Have a look at the Chipwhisperer Jupyter notebook "Lab 3_3 - DPA on Firmware Implementation of AES" for some hints on how to improve it
- You don't have a limit on the number of traces, however we expect that you never need more than 10K power traces
- [SBOX 1] best_key = 0x27

Step 5: CPA - Correlation Power Analysis

As you have noticed in the previous step, DPA is quite noisy and you need a lot of traces to extract the key. With the help of math you can get much better results. Try to modify your script to perform CPA. CPA can be simplified as "DPA on all the SBOX output bits". For doing this it's better to talk about correlation. We suggest to use the Pearson Correlation Coefficient with the goal to find the best correlation between X and all the Y arrays where:

- X[i] = hamming weight of the S-Box output of the i-th trace.
- Y[i][j] = trace[j].sample[i];
- len(X) == len(Y[i]) == n_traces

A high correlation between X and Y[i] means that in every trace, at sample i, the hamming weight of the sbox output matches with the power consumption of that sample. In other words, the current guess key was always able to produce a correct power estimation, thus making it a good candidate. If you implement this approach you will see that the required number of power traces and the noise are drastically reduced.

Task 5

Deliverables:

- Your script to perform DPA to recover K1.
  IMPORTANT: Your script must do everything from the capture to printing the best K1 candidates! The only argument of your script is the number of traces to capture. (so the script should be executed as ./cpa <n_traces>).
    - Full score will be given only if the entire K1 is retrieved by using only the best candidate

- A report describing your **results**, **how the attack works** and **how to run the script**. Notice that if we cannot run your script you will get 0 points for this part!
- **You will need the full DES key for next week's assignments! All the 8 parity bits of the 64 bit key are set to 0 for simplicity**. For that you need to revert the DES steps from K1 to the initial full key.

Hints:

- Jupyter Notebook of CW: “Lab 4_2 - CPA on Firmware Implementation of AES”
- To verify you recover fully the DES key you can use CyberChef

Step 6: improving CPA (Bonus)

With CPA you can get quite good results!
Can you find a way to improve this analysis and leak K1 with less power traces?