4 Side Channel Analysis Attacks (Part 1)

Statistical Analysis of Information Leakage

A Introduction

Electronic devices are prone to leak information about internal processes through side channels such as supply current, electromagnetic (EM) radiation, or timing. A knowledgeable attacker can take advantage of this fact by observing and analyzing differences in patterns in the side channels. An attacker may use this information to gain insight into what operation is being performed by the device, or even what data is being processed at a particular time. This is possible because, as the device performs different operations, or the same operations on different data, there will be different transistors switching on and off, different values passed between circuits along busses, and so on. Side-channel Analysis (SCA) attacks exploit this physical information leakage from electronic circuits to extract secret information. SCAs take advantage of the difference in those patterns and, through measurements and analyses, enable an attacker to extract information that can help them to better understand the circuit behavior. These attacks can be relatively inexpensive and easy to mount on a target device.

B Background

Side-channel information leakage testing attempts to identify cases where side-channels parameters, like power consumption or timing, clearly vary with respect to some hidden internal parameters, such as the data being processed. For example, given two functions $f(x, y)$ and $g(x, y)$, if one of these functions performed a complex operation, e.g. $x^2 + y^2 + xy$, while the other performed a simple operation, e.g. $x \oplus y$, it should be fairly easy to tell which function is doing what operation based on the time it takes to return. This is an example of a *timing* side channel. In a *power* side channel, we can look at how the power consumption in the chip might vary depending on what data the chip is processing at a given time. In particular, wide operations, such as a 8-bit `XOR`, would consume *slightly* (but measurably) more power when there are more 1s on the output. For example, $00110101 \oplus 10001010 = 10111111$, which has seven 1s, would consume slightly more power than $00110101 \oplus 10110100 = 10000001$ which has only two 1s on the output. Various techniques, which will be the subject of subsequent labs, will introduce methods for exploiting these tiny variations for recovering the secret key.

Before delving into more complex techniques for key recovery, we can try to observe if there is a significant difference when the device performs different functions on the same data, or when the device performs the same function on different data. This is especially useful for encryption, where the same function (e.g. AES) is being performed. For example, given a plaintext and the key, we can try to observe if there is a significant difference when the device encrypts two different strings, e.g. "Hello!" and "Hell0!", or encrypts the same string using two different encryption keys. Note the use of the word "significant" - because we will be using statistical analysis to help us. In particular, **Test Vector Leakage Assessment (TVLA)** is a standard baseline approach to determine side-channel vulnerability by intentionally varying an internal parameter and measuring whether or not a change is detectable. This approach makes use of a statistical hypothesis test (SHT) - Welch's t-test - to determine whether the two sets of recordings have identical means when standardized inputs are used. These inputs are defined in the original TVLA specification, which can be found here .

The recording of the side channel as it varies with time, over the course of a single operation, e.g. encrypting a single 128-bit plaintext using AES, is said to be a *trace*. An example trace is shown in Figure 1. To conduct the test, we will need to collect several hundred traces (or more) using different inputs.

Figure 1: Example of power trace acquired during a single AES operation.

The t-test tests the hypothesis considering whether or not two independent groups have an identical mean; a typical application of this test is to determine the mean equivalence of two populations. Before running the t-test, there are some important details to take into consideration. First, we must formally define the null hypothesis, H_0, and the alternative hypotheses, H_a. Second, we must specify the significance level, α, which will depend on the test requirement. Finally, we evaluate the null hypothesis based on the computed probability value (P-value) from a data sample. The P-value is the probability that the sampled data would occur if the null hypothesis is true. The null hypothesis generally states that there is no *statistically significant* difference between the two populations. In hypothesis tests, the typical level of significance used is 0.05(5%), which means there is a 5% probability of rejecting a true null hypothesis. Further, the 5% significance level means that there is a 95% confidence level, which translates to a 95% probability of *not* rejecting a true null hypothesis. The statistical significance level is expressed as a P-value, which lies between 0 and 1. When $P \leq 0.05$, it suggests that there exists statistical significance which contradicts the null hypothesis – meaning we *reject* the null hypothesis in favor of the alternative hypothesis. When $P > 0.05$, it suggests there is no statistical significance, which also means it fails to reject the null hypothesis. As an example, we can define the null hypothesis as, "There is no significant difference between the two different sets of collected EM signals," considering the significance level, α as 5%.

What does this mean for SCA? If we encrypt the same text with two different keys, or use one key to encrypt two different plaintexts, we would expect there to be some difference in the power consumed by the chip during this operation. However, these changes might be very subtle, and we may not be able to see them visually on a graph. The signal can also be *noisy*, which could confound the analysis. In either case, we can use statistical tools to help us determine whether there is significant information leakage from our chosen side channel. Importantly, evidence of leakage is *not* evidence that the key can be recovered with further analysis, but it is a starting point when evaluating the efficacy of an SCA countermeasure.

Reading Check

1. What are some examples of physical parameters that an attacker can leverage in an SCA attack?
2. Why does an unprotected electronic device (one without any countermeasures in place) leak information through side channels?
3. What does the p-value represent in a statistical hypothesis test, such as the t-test?

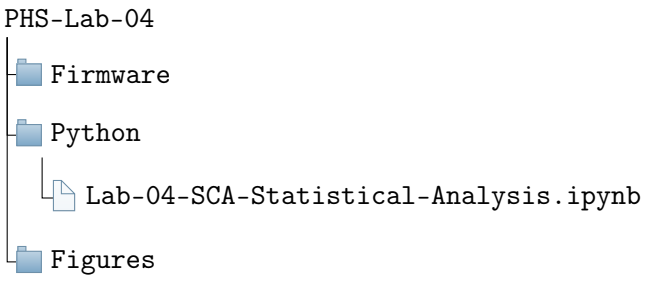
C Hardware and Software Tools

For this lab, you will utilize the ChipWhisperer (CW) Nano board. The required software tools are PHS-VM (Appendix C), and the provided lab module files stored in "PHS-Lab-04.zip." The PHS-VM includes ChipWhisperer v5.5 for Python v3.8-3.10 that provides an easy-to-use interface with the CW. Also, the PHS-VM includes a directory called "cw-base-setup" with all CW setup scripts (do not change) and example Firmware. In this lab, we will use "cw-base-setup/simpleserial-base" as a starting firmware template to program the CW Nano board.

Note: you only need one (1) CW Nano board for this lab. You are not required to repeat this procedure on both boards. However, you are encourage to use both CW to distribute and verify each other's work.

D Getting Started

First, download and complete "PHS-Tutorial-03-CW" (Introduction to ChipWhisperer); see the recorded lecture for hints and details. If you are unable to complete these two tutorials, see the instructor or TA(s) for assistance.



E Lab Assignment

After completing the tutorial, you should have a basic understanding of the CW Nano platform. We will use the platform in this lab to compile custom firmware for the target microcontroller, acquire power traces, and use Python to analyze the signals to perform a kind of TVLA.

Part A: Firmware

1) Under "cw-base-setup/simpleserial-base" directory, make a copy of the "simpleserial-base" files (simpleserial-base.c and the makefile) and move them into your "PHS-Lab-04/Firmware" directory.

2) Under "PHS-Lab-04/Firmware" directory, rename the "simpleserial-base.c" copy to "tvla-test.c", and update the makefile accordingly:

   a) lines 33 and 37 of the "makefile" with the updated file name.

   b) because the files moved location, lines 45 - 49 replace with:

Listing 1: "makefile": replace lines 45 - 49

```
#Path to Base/Setup Firmware Directory (i.e., cw-base-setup)
FIRMWAREPATH = ../../cw-base-setup

#Add simpleserial project to build
include \$(FIRMWAREPATH)/simpleserial/Makefile.simpleserial

#Add other relevant components project to build
include \$(FIRMWAREPATH)/Makefile.inc
```

3) Open the "tvla-test.c" file and, using the function `get_pt` as a template, create two new functions, `tvla_test_A` and `tvla_test_B`. Add these two function callbacks in the `main` method, using simpleserial parameters "a" and "b", respectively.

4) Modify the `tvla_test_A` function to perform a product across all 16 elements of the input array, and return the final value. Similarly, modify the `tvla_test_B` function to XOR all 16 elements of the input array.

   For the graduate students, we also require that you:

5) **Graduate Only:** Under the same "tvla-test.c" file, create and modify another function called `tvla_test_C` as you did before. Add this function callback in the `main` method, using simpleserial parameter "c". Similarly, this function should preform a modular multiplication across all 16 elements of the input array with modulo 0xFB (= 251). In other words, pt[16] is your array with 16 hex values and M is the modulo, then the modular multiplication between them is equal to: $(pt[0] * pt[1] * pt[2] * ... * pt[15])$ `mod` M

   This function by itself may be computational heavy for your CW. It is attempting to compute the modulo of a very large number. But there are techniques for simplify it. Hint: look up properties of modular multiplication.

Part B: Python

1) Under the "Python" directory, open the "Lab-04-SCA-Statistical-Analysis.ipynb" script. Copy over the relevant connection code from "PHS-Tutorial-03-CW" (Introduction to ChipWhisperer), including imports for chipwhisperer, matplotlib.pyplot, numpy, and scipy.stats.

2) Modify the Jypyter Notebook's Python code so that it compiles the C program you wrote, then program the target device. Confirm your C code is working by writing equivalent functions in Python (you can define these in their own cells in the notebook), sending several different values (at least 3) to the target via serial, encoded in a bytearray, and cross checking the results.

3) Once you have confirmed functionality, use CW Nano to acquire 32 traces for two different data. For each of the 32 iterations, send the same test data each time to `tvla_test_A`, and append this to a list. Repeat with a second data. Convert the two lists to numpy arrays. Average (sample-wise) the traces for each data, and plot them using pyplot. Subtract the two traces to generate a differential trace; plot this separately. What do you observe? Compute and report the mean and standard deviation for the three signals (average data A, average data B, and differential trace).

4) Repeat the experiment for `tvla_test_B`, plot the resulting average traces, and generate and plot the differential trace. What do you observe this time? Again, compute the mean and standard deviation for all three signals.

5) Using scipy, perform Welch's t-test (note the optional equal_var parameter!). Note also that this function returns a pair of values, t and p. Obtain the t and p values for the original pair of 32 traces from Part B, Step 3. Trim this down to the first 250 samples for each 32 traces, e.g. `trace_A1[`i`][0:250]` and `trace_A2[`i`][0:250]` (for $i = 0$ to 31), and plot the resulting array of 250 t values. Repeat this for the traces from Part B, Step 4. What percentage of the reported p values are considered "significant" and result in rejecting the null hypothesis?

For the graduate students, we also require that you:

6) **Graduate Only:** Repeat all these procedures with `tvla_test_C`.

F Badges and Achievements

This experiment involves **firmware**, **serial communication**, and **data visualization**. By successfully completing the lab you can unlock the following achievements:

- C1: Firmware Programming in C
- COM2: Serial Communication
- STAT4: Statistical Analysis

Depending on the demonstrated skill, each achievement will be further divided into bronze, silver, and gold levels.

G Deliverables

Follow all report guidelines as detailed in the Appendix A. The following result(s) must be included in the report:

- Answers to the reading check questions
- 3 plots from Part B, Step 3.
- 3 plots from Part B, Step 4.
- 2 plots from Part B, Step 5.
- Percentage of p-values from Part B, Step 5 which indicate statistical significance at the 5% threshold (i.e. are below 0.05).
- Mean and standard deviation of each average traces, differential trances, and t-values.
- **Graduate students** need to get 4 more plots from Part B, Step 6. Also, the percentage of p-values from Part B, Step 5 which indicate statistical significance at the 5% threshold.

Submit your report along with all code. All analysis code should be runnable and should produce the output submitted with your report. Files, including your C code and Python notebook, should be downloaded and submitted to Canvas along with your report files.