5 Side Channel Analysis Attacks (Part 2)

Finding Patterns in the Power Side Channel

A Introduction

In Lab 4, you experimented with taking measurements of power consumption from a target device and analyzing the signals to see if any data is "leaking" through side channels. In that example, the same operations were performed every time, so you used statistical tests to check if there was any discernible impact on the power consumed.

In this lab, we will take a slightly different approach and look not just at how much power is consumed, but *when* it is consumed. We will look for patterns in the signal which can give us information on the algorithm running inside the chip. We will then use this timing information to make an inference about the information being processed using a simple password checking routine.

B Background

Timing attacks will often exploit optimizations in the code. Take for example the following password checking routine, which compares byte-for-byte each character in the two arrays, but stops comparisons when a mismatch is found:

Listing 1: Password Checker v1.0

```
char real_password[] = "passWord";
char test_password[] = "password";

int password_wrong = 0;

for(uint8_t i = 0; i < sizeof(real_password); i++)
{
    if (real_password[i] != test_password[i]) // mismatch
    {
        password_wrong = 1;
        break;
    }
}

if (password_wrong)
    printf("WRONG PASSWORD");
else
    printf("ACCESS GRANTED");
```

Once a mismatch is detected, the processor will break out of the loop and will continue on to other tasks. Since it immediately gives feedback to the user, this could be exploited by a brute force attack, simply timing how long it takes to get the error message back from the system. To counter this, you may try implementing a random delay before letting the user know the password is incorrect, for example:

Listing 2: Password Checker v2.0

```
char real_password[] = "passWord";
char test_password[] = "password";

int password_wrong = 0;

for(uint8_t i = 0; i < sizeof(real_password); i++)
{
    if (real_password[i] != test_password[i]) // mismatch
    {
        password_wrong = 1;
        break;
    }
}

if (password_wrong)
{
    int wait = rand() % 12345;
    for(volatile int delay = 0; delay < wait; delay++){
        ;
    }
    printf("WRONG PASSWORD");
}
else
    printf("ACCESS GRANTED");
```

Even so, with a timing attack on the power side channel, we should be able to see how many iterations took place in the main password checking loop before breaking out. In this case, the password is known, so we can intentionally send the correct or incorrect bytes to learn more about how it handles password checking. Figure 1(a) shows an entire power trace for a correct password entry. Password checking is taking place within the first 100 samples, after that you can see the power signature while the processor is in an infinite loop. Figure 1(b) shows three overlaid power traces. There is a clear distinction between the three traces between 0 and 100 samples while the password is being checked. All traces start off the same but eventually they diverge. The black line is the trace for no correct bytes, the red line has one correct byte, and the blue line has two correct bytes. Zooming into (Figure 1(c)), the pattern becomes clear. Notice the very unique shape of the power spike at sample 35, 46, and 57. It would seem that this spike could be used as a kind of "signature" to determine how many bytes are correct: if there is a spike at sample 35 (e.g. $trace[34] > 0.4$), then we have 0 characters correct. For each subsequent character that is correct, that same spike is shifted by $11 \times i$, where i is the number of correct characters. Note that on your particular board and depending on the compiler used, the timing, shape and amplitude of the spike may differ slightly. For example, on another board, the first spike always occurs at sample 27 at a different level (e.g. $trace[27] > 0.0$) with successive "correct" spikes occurring every 13 time steps.

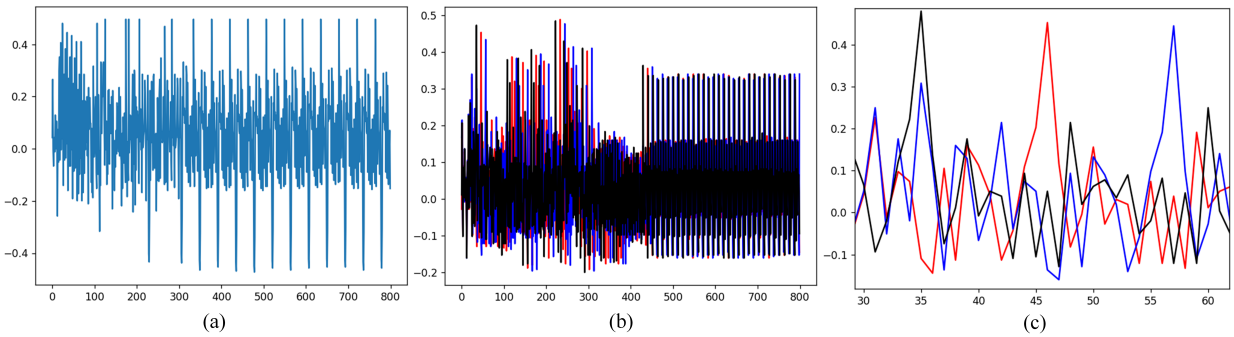


Figure 1: (a) Sample of a power trace recorded from a correct password. (b) Plot of three different passwords, one with no correct characters, one with a single correct character, and one with two correct characters. (c) Zoomed in view from (b) showing unique power spikes for guesses with 0, 1, or 2 correct characters.

Reading Check

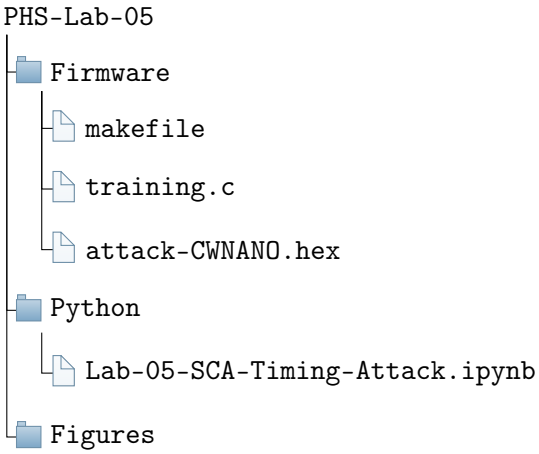
1. How does a timing attack infer information from the physical chip?
2. What do timing attacks often exploit?
3. What are some ways to defend against timing attacks?

C Hardware and Software Tools

For this lab, you will need the ChipWhisperer Nano or equivalent board, as well as a ChipWhisperer VM set up as in the previous lab.

D Getting Started

Download and open the PHS-Lab-05 directory, which contains 3 folders: Firmware, Python, and Figures.



The **Python** directory contains a Jupyter Notebook which walks you through the attack.

The **Firmware** directory contains the scripts needed to program your CW's target: "training.c", "attack-CWNANO.hex", and "makefile". The makefile is needed to compile the "training.c" firmware and generate the "training-CWNANO.hex" file. The "training.c" contain the password checkers v1.0 and v2.0 as described before. The "attack-CWNANO.hex" file was compiled from "training.c" but with a different secret password. You will use the training files to practice and get yourself comfortable with the attack, then use the "attack" file to perform the attack on an unknown password.

The **Figures** directory is where you should save all your waveforms (i.e., plots) generated from the notebook.

Lab Assignment

Begin by opening the Lab-05-SCA-Timing-Attack notebook. Follow the instructions to compile "training.c" and program the target with the respective training-CWNANO.hex file. You should be able to send a password attempt to the device using `target.simpleserial_write()` and receive a response. There are two implementations of the password checking routine, corresponding to the **Naïve Password Checker v1.0** and **Naïve Password Checker v2.0** (simpleserial command `a` and `b`, respectively). Hint: the correct password (for this portion) is `USFCSE`. Try sending a few different passwords. Try timing the response time using `time()`; is there a difference when you enter 1 character correct (e.g. U01234) versus all 6 characters correct when using the original password checker? What about with v2.0? You may need to record multiple trials and average them together to get a more accurate measurement.

Part A: Training Password Attack

1) Under the "Python" directory, open the "Lab-05-SCA-Timing-Attack.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 firmware "training.c", then program the target device in "Section 1".
3) Follow the given Jupyter Notebook Lab-05 code and make sure you understand the password checker interface and how the attack works. You may start from the given notebook template and fill in blank sections with your own code.
4) Use the scope and capture some traces for 0, 1, 2, 3, 4, 5, and 6 characters correct. Do you see any repeating patterns? Try subtracting two traces (one with 1 character correct, one with 5 characters correct). Does that give you any more information?
5) Implement a single character attack function on "Firmware/training.c". You should read this firmware file to better understand how the password checker v1 and v2 works. The single character attack should work as follows:
    a) If given a reference password with the first k correct characters, find through power SCA timing attack the $(k+1)^{th}$ correct character (if any).
    b) Howevever, if there is no $(k+1)^{th}$ correct character, then return `False`. In other words, the input reference password is already the correct password.

6) Once successful, implement a full password attack function on "Firmware/training.c" to find all the correct characters. You can reuse the previous single character attack function for this step. Save outputs showing the waveforms you have identified as belonging to the password checking routine, and clearly mark areas where 1, 2, 3, etc. characters of the password have been correctly guessed. The attack should be *fully automated*.

7) Perform the full password attack for both password checkers on "training.c": Password Checker v1.0 and v2.0

Part B: Secret Password Attack

1) Once you are comfortable with the attack, load the "attack-CWNANO.hex" file into the target board. This time the *password* and its *length* are not known. It does however have only English alphabetical (both upper and lowercase). Using this information and your experience from the "training.c" file, write a script which will automate the attack and break the password using the timing and the power side channel.

2) Implement the necessary functions to mount the attack. Save outputs showing the waveforms you have identified as belonging to the password checking routine, and clearly mark areas where 1, 2, 3, etc. characters of the password have been correctly guessed. The attack should be *fully automated*.

   Hint: "attack-CWNANO.hex" was compiled from the same "training.c" file but with a different unknown secret password. You need to find this secret password.

3) Perform the full password attack for both password checkers on "attack-CWNANO.hex": Password Checker v1.0 and v2.0

Once successful, modify the password checking routine to prevent this kind of timing attack:

Part C: Password Checker Countermeasure

1) Resuse the same "training.c" file. Open it and make a copy of either function: `password_checker_v1` or `password_checker_v2` (whichever you prefer). Rename the new function to `password_checker_v3`.

2) Modify the `password_checker_v3` with a countermeasure that prevents your full password attack.

3) Add this new function callback in the `main` method, using simpleserial parameters "c".

4) Retry the same full password attack on this improved password checker. It should thwart your attack in previous parts. Save outputs showing the waveforms you have identified as belonging to the countermeasure password checking routine.

E Badges and Achievements

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

- CRYPTO1: Side Channel Analysis
- C2: Firmware Programming in C
- PLOT4: Data Visualization
- STAT4: Statistical Analysis

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

F Deliverables

Submit your report along with all code. Set up your experiments along the guidelines described above, and organize your findings into a report according to the guidelines detailed in the Appendix A. The report should also contain the following:

- Answers to the reading check questions.
- Waveforms from:
  - Part A, Steps 4, 6, and 7
  - Part B, Steps 2 and 3
  - Part C, Step 4
- Screenshots of the output of your full password attacks from:
  - Part A, Steps 6 and 7
  - Part B, Steps 2 and 3
  - Part C, Step 4