Lab 4: Side Channel Analysis Attacks Part 2: Power Consumption & Timing Analysis

Igli Duro
Department of Computer Science & Engineering
University of South Florida
Tampa, FL

I. Introduction

For this lab, we evaluate timing as an attack vector for SCA attacks. Timing attacks are done in tandem with power consumption attacks to look at when exactly more power is being consumed. From this an attacker can look for patterns in the power consumption to try and infer how information is being processed or what an algorithm is potentially doing. This is possible as algorithms are usually written in optimized ways. And these optimizations may result in security flaws. For example, a function may exit early if some passed argument is incorrect. While the correct information may take longer to return as more operations could be running on it. As part of this analysis we'll be looking at a password checking algorithm and see if we can identify patterns to gain access without knowing the correct password beforehand. Then modifying it to prevent our attack.

II. Methods

This lab provided a ChipWhisperer Nano board which used a virtual machine and a Juptyer notebook to interface with it. We were given a "Starter Notebook" with some steps already provided and some firmware files to program the CW Nano board. These files are Lab-05-Attack.hex, Lab-05-Training.hex, Lab-05-Training.c, and makefile. The hex files would be used to program the CW Nano in a training mode where the password is known for analysis purposes and an attack mode where the password is not known and we have to mount an attack ourselves. In our Starter Notebook we first were in training mode. As said the password was known (USFCSE) and due to that were were able to capture and graph traces of the password checking algorithm using passwords with varying levels of correctness. This was with 0, 1, 2, 3, 4, 5, and all characters being correct. By graphing it we were able to find areas of interest. These were "spikes" of power consumption. We knew that the algorithm checked the password byte by byte and returned early when a character was incorrect. So noticeably, a password attempt with 0 characters correct had a spike earlier than one with 1 character correct. This trend stood for the other attempts too and not surprisingly the correct password had the latest spike. Using this we were able to see a trend in spikes per character correct.

From here we used the spike in power, and timing it occurred to create an algorithm for testing characters of the alphabet to find a match. This algorithm is called "findLetter" and takes in the approximate range a spike occurred, as in multiple runs it may vary by 1 or 2 samples, along with the characters of the password found to far, the position of the character to be found, and a list containing the alphabet. After the letter was found it would be returned and then saved to a string containing all characters found so far. After being able to find a single correct character we then automated the process for finding the entire string for the correct password. This automated process would build a password character by character and then pass it to the password checking algorithm until access was granted. Once we were done with training mode we then used our algorithm to mount an attack and crack a previously unknown password. Results from this were then output to the terminal and graphed.

Afterwards we used a different password checking algorithm that aimed to fix this timing vulnerability by introducing a random delay before a incorrect password attempt is returned. The idea behind this is to add a variable that would scatter the power spikes so that a pattern could not be formed. Results from this were also graphed.

III. Results

A. Reading Check

1. *How does a timing attack infer information from the physical chip?*
Timing attacks can infer information on a chip by being able to see when more power is consumed. From there and with enough testing a pattern can be formed and one can begin to infer why and what's happening in an algorithm.

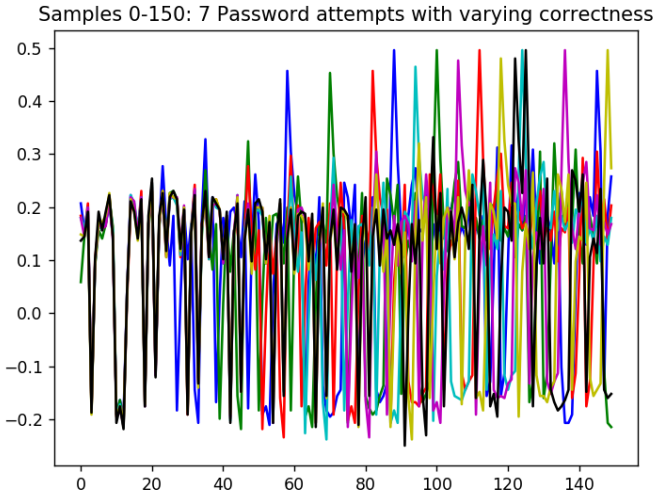
2. *What do timing attacks often exploit?*
Timing attacks often exploit optimizations in the code. For example if there is a loop running under a condition and that condition is affected by user input, then that input can cause the program to return earlier or later, as if a condition for a loop is broken or met then there isn't any benefit of running redundant operations if an answer is already determined.

3. *What are some ways to defend against timing attacks?*
Some ways to defend against timing attacks is to make code more uniform or random. If there are branches in code with conditional statements, then you can make those branches have the same number of operations so that regardless of which taken, the time taken would be similar. This may need some redundant operations but make it more secure to timing attacks. Making the code more random would be

by introducing a varied amount of delay by using a random number generator and a wait() function. By doing this it makes it harder to find patterns as on subsequent runs results for timing would appear different.

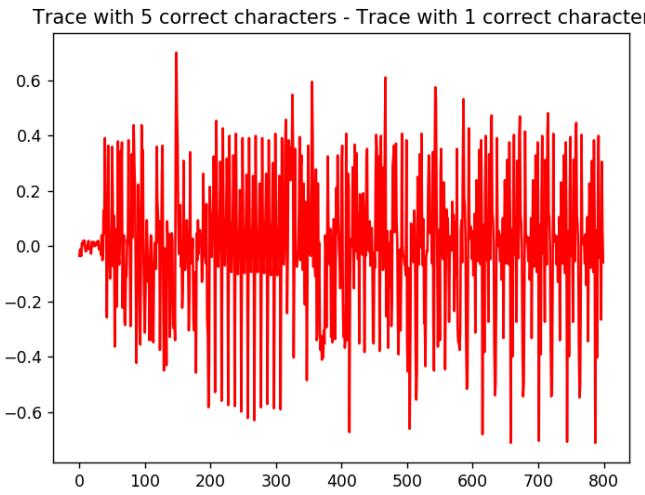
B. Analysis of Results

The graphs below show the results of samples from the captured password traces. The first one with 7 password attempts were from training with a known password. The blue trace was with 0 correct characters, green was 1, red was 2, cyan was 3, magenta was 4, yellow was 5, and the black trace was with the correct password. Spikes greater than 0.4 were our area of interests and we can see a fairly identical trend in spikes with uniform lengths. This became the basis for our automated attack as the difference in sample number between spikes became our ”step” for checking the next characters for the next spike. Applying this to the graph for 12 password attempts which was done on an unknown password we can see the same trend, in this graph red was with 1 character correct, blue was 2, green was 3, cyan was 4, magenta was 5, yellow was 6, orange was 7, purple was 8, brown was 9, pink was 10, grey was 11, and black was the correct password with 12 characters. Next is our output, as can be seen our algorithm for building a password character by character was successful, showing what letter and sample number it was found on. The uncovered password was ”TimingSCAftw”. Lastly is the graph using the same 12 passwords but on a password checking algorithm that introduces a random delay before returning an incorrect password. While we can still identify power spikes the order the they come in and the time they occur are quiet varied. We'll discuss this further in the next section.



IV. Discussion

As mentioned in Section III, running the password checking algorithm with a random delay before returning resulted in varied amounts of spikes. Unsurprisingly, while our algorithm was able to find the unknown password using analysis built upon power spike timing per correct character, it was unsuccessful at trying to so again with the varied delay version of the algorithm. Not only were power spikes not in the same order, but when the occurred was different as-well. Subsequent runs of this would produce different results making finding the password on power-consumption and timing alone quite the improbable challenge. Brute forcing is still technically an option for anyone that wants to try 52 to the power of 16 possible combinations.



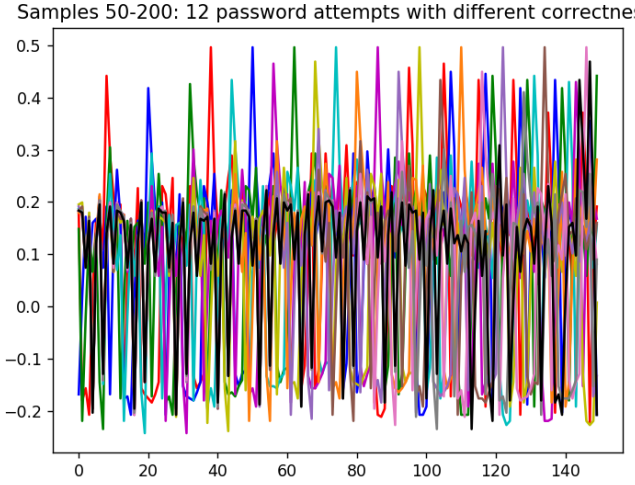
V. Conclusion

In this lab, power and timing analysis were used to identify information being processed in a password checking algorithm. We started off with a training set using a known password and were able to uncover power and timing patterns based off the number of correct characters. This information was then used to build a algorithm to build a password character by character by looping through the alphabet and finding matches for the timing and power at a given position. We were then able to use this algorithm to uncover a password that was unknown to us, which turned out to be ”TimingSCAftw”. Afterwards we used a modified password checking routine that introduced variable delay for incorrect attempts which thwarted our algorithm as we could no longer identify a pattern.

```
Found letter:  T  on trace # 70

Found letter:  i  on trace # 82
T
Found letter:  m  on trace # 94
Ti
Found letter:  i  on trace # 106
Tim
Found letter:  n  on trace # 118
Timi
Found letter:  g  on trace # 130
Timin
Found letter:  S  on trace # 142
Timing
Found letter:  C  on trace # 154
TimingS
Found letter:  A  on trace # 166
TimingSC
Found letter:  f  on trace # 178
TimingSCA
Found letter:  t  on trace # 190
TimingSCAf
Found letter:  w  on trace # 197
TimingSCAft
TimingSCAftw
TimingSCAftw
ACCESS GRANTED
```

Samples 50-200: 12 password attempts with different correctness



Samples 50-200: Same 12 passwords but with a random delay