CPE 426 - Introduction to Hardware Security

Lab 2 - Power Side Channel

Presented to:
Dr. Stephen R. Beard
California Polytechnic State University, San Luis Obispo

Written by: Hardware InsecuritEE
Qingyu Han, Weston Keitz, Nathan Jaggers

Fall 2023

Table of Contents

I. Questions

Question 1

Include the code from the "Lab 3_3 - DPA on Firmware Implementation of AES (MAIN)" cell for the "AES Guesser - All Bytes" piece.

```
def aes_secret(inputdata):
    secret_key = 0xEF
    return aes_internal(secret_key, inputdata)

# ###################
# START SOLUTION
# ###################
sbox = [
    # 0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76, # 0
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0, # 1
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15, # 2
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75, # 3
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84, # 4
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf, # 5
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8, # 6
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2, # 7
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73, # 8
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb, # 9
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79, # a
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08, # b
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a, # c
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e, # d
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf, # e
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16  # f
]

def aes_internal(inputdata, key):
    return sbox[inputdata ^ key]
# ###################
# END SOLUTION
# ###################

#Simple test vectors - if you get the check-mark printed all OK.
assert(aes_internal(0xAB, 0xEF) == 0x1B)
assert(aes_internal(0x22, 0x01) == 0x26)
print("✔️ OK to continue!")

#
# Perform the capture, resulting in trace_array and textin_array of 2500 traces.
#

SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEARM'
CRYPTO_TARGET='TINYAES128C'
SS_VER='SS_VER_1_1'

%run "../../Setup_Scripts/Setup_Generic.ipynb"

%%bash -s "$PLATFORM" "$CRYPTO_TARGET" "$SS_VER"
cd ../../../hardware/victims/firmware/simpleserial-aes
make PLATFORM=$1 CRYPTO_TARGET=$2 SS_VER=$3
```

```
cw.program_target(scope, prog,
"../../../hardware/victims/firmware/simpleserial-aes/simpleserial-aes-{}.hex".format(PLATFORM))

from tqdm import tnrange
import numpy as np
import time

ktp = cw.ktp.Basic()
trace_array = []
textin_array = []

key, text = ktp.next()

target.set_key(key)

N = 2500
for i in tnrange(N, desc='Capturing traces'):
    scope.arm()

    target.simpleserial_write('p', text)

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

    response = target.simpleserial_read('r', 16)

    trace_array.append(scope.get_last_trace())
    textin_array.append(text)

    key, text = ktp.next()

assert(len(trace_array) == 2500)
print("✔️ OK to continue!")

%matplotlib notebook
import matplotlib.pylab as plt
plt.plot(trace_array[2000])
plt.plot(trace_array[2400])

print(textin_array[0])
print(textin_array[1])

numtraces = np.shape(trace_array)[0] #total number of traces
numpoints = np.shape(trace_array)[1] #samples per trace

# ###################
# Add your code here
# ###################

import numpy as np
mean_diffs = np.zeros(256)

guessed_byte = 0

for key_byte_guess_value in range(0,256):

    one_list = []
    zero_list = []

    for trace_index in range(numtraces):
        input_byte = textin_array[trace_index][guessed_byte]

        #Get a hypothetical leakage list - use aes_internal(guess, input_byte)
        hypothetical_leakage = aes_internal(key_byte_guess_value,
textin_array[trace_index][guessed_byte])

        #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)
    max_diff_value = np.max(abs(one_avg - zero_avg))

def calculate_diffs(guess, byteindex=0, bitnum=0):
    """Perform a simple DPA on two traces, uses global `textin_array` and `trace_array` """

    one_list = []
    zero_list = []

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

        #Mask off the requested bit
        if hypothetical_leakage & (1<<bitnum):
            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)
    return abs(one_avg - zero_avg)

%matplotlib notebook
import matplotlib.pylab as plt

plt.plot(calculate_diffs(0x2B), 'r')
plt.plot(calculate_diffs(0x2C), 'g')
plt.plot(calculate_diffs(0x2D), 'b')

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]

for subkey in tnrange(0, 16, desc="Attacking Subkey"):
    # ###################
    # Add your code here
    # ###################
    mean_diffs = np.zeros(256)
    max_diff_value = []
    for key_byte_guess_value in range(0,256):

        one_list = []
        zero_list = []

        for trace_index in range(numtraces):
            input_byte = textin_array[trace_index][subkey]

            #Get a hypothetical leakage list - use aes_internal(guess, input_byte)
            hypothetical_leakage = aes_internal(key_byte_guess_value,
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)
        max_diff_value = np.max(abs(one_avg - zero_avg))
        mean_diffs[key_byte_guess_value] = max_diff_value
    key_guess.append(np.argsort(mean_diffs)[::-1][0])

print(key_guess)
print(known_key)
```

<u>Question 2</u>

In your own words, briefly describe how this attack works, i.e. what is actually happening in the code for 1.

The basics of the attack originate from exploiting knowledge from a leaked single bit of data. In Lab 3_2, we could brute force a signal bit of data by performing the s-box operation on a single byte of input data, and a single byte key. Each bit is determined by counting the matching s-box hash output to all 256 possible inputs (0x00 to 0xFF). The result with the most matching bits is the result for determining the key of the AES encryption.

Lab 3_3 builds off of the exploitation concepts in Lab 3_2 by performing a differential power analysis (DPA) attack to gather the bit information based on the power consumption of the device to guess the AES encryption key. If the result of the hash is a 1, there should be higher consumption than if the output is a 0. On the ChipWhisperer board, 2500 power traces are collected from an AES encryption. Of the 16 byte key, each byte is brute forced through the same concept as in Lab 3_2 by testing all 256 possible combinations. Through the trace of the LSB of the s-box output for each key will all traces, any value with a 1 as the LSB would place the voltage value in the ones list and the 0s in the zeros list. The correct byte will have the highest differential of power values of the average of the ones list and the zeros list. This process is repeated for all bytes.

Question 3

What are some potential defenses against this kind of attack?

- **Randomized Execution**: Randomize the order of operations for the AES encryption process to make it harder for attackers to correlate power consumption with specific operations order.
- **Constant Power Consumption**: Implementing hardware or firmware that ensures constant power consumption regardless of the output being a 1 or 0.
- **Noise Addition**: Introducing artificial noise in power traces at the output can make it difficult for attackers to identify the state of the output.
- **Physical Shielding**: Shielding the cryptographic device physically to prevent easy access to power analysis.
- **Blow the whole fucking thing up:** If you want it to be secure as possible, make it blow up if probed or make it not exist.

Assignment Review

<u>Question 1</u>

Did completing this lab teach you anything or serve to reinforce something we learned in class?

In Week 5, we covered the paper "In the blink of an eye: There goes your AES key" which could extract the AES key from pipeline emission analysis. It was helpful to read about a specific case of encryption attacks from side channel analysis then perform our own experiment in class.

<u>Question 2</u>

Do you think the lab was worth doing?

Yes, this lab offers practical hands-on experience with understanding and implementing a DPA attack, which is essential for comprehending the importance of side-channel security. Understanding theoretical attacks and seeing how they are executed in the real-world is invaluable for designing cryptographic systems. This lab also emphasized that no system is impervious to attacks, which is crucial for developing a mindset that gears towards continuous improvement!

<u>Question 3</u>

Roughly estimate how long this lab took you.

This lab took about 6 lab sessions for our group. The first one or two sessions were used to get familiar with the environment and set up to conduct these labs. This could probably be reduced down to one lab session if in conjunction with the in class demo on how to start up the Jupyter Notebook, there was a video tutorial or guide document to show the same steps in case students get lost. Also in the same video or document it could rehash some of the same info in the Jupyter Notebook about how there is a hardware and simulated versions on proceeding with the lab. That being said these labs were not bad or hard to follow at all, one just would need the time to delve into the material. From there, most sections in the lab could be completed in one lab session (1.5 - 2 hours).

<u>Question 4</u>

Do you think the class should have included additional labs or attacks from the Chip Whisperer notebooks?

The Chip Whisperer is a very cool device and has some very interesting labs. They are also laid out in a way where the labs are not too difficult and you can learn a lot from them. I am sure everyone would have fun being able to spend some more time with the Chip Whisperer with labs like the Correlated Power Analysis or the fault labs like Clock Glitching. There are a few caveats with this though and that is that there is only 1 Chip Whisperer per group so you have to keep handing off the device from one person to another (We did not find a way to sync up notebooks through GitHub). Then there is the time aspect to it because along with this there is Lab 1 and the Hardware Trojan Project. So it is hard to imagine being able to fit in more labs into the quarter without replacing some of the already in use content. We thought the PUF lab was pretty cool and the Trojan Project is a fun final exercise to cap off the class.