In [1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

CS529-HWS: Hardware Security - Lab 3

Aneesh Kandi | [kandia@purdue.edu]

Connecting to the ChipWhisperer Nano

In [ ]:
```
SCOPETYPE = 'CWNANO'
PLATFORM = 'CWNANO'
CRYPTO_TARGET = 'TINYAES128C'
SS_VER='SS_VER_1_1'
```

The following code will build the firmware for the target.

In [ ]:
```
%run "../../Setup_Scripts/Setup_Generic.ipynb"
```

```
INFO: Found ChipWhisperer😍
```

In [ ]:
```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET" "$SS_VER"
cd ../../../firmware/mcu/simpleserial-aes
make PLATFORM=$1 CRYPTO_TARGET=$2 SS_VER=$3 -j
```

```
Building for platform CWNANO with CRYPTO_TARGET=TINYAES128C
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Blank crypto options, building for AES128
arm-none-eabi-gcc (15:10.3-2021.07-4) 10.3.1 20210621 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOS
E.

mkdir -p objdir-CWNANO
.
Welcome to another exciting ChipWhisperer target build!!
.
.
.
Compiling:
Compiling:
.
Compiling:
-en     simpleserial-aes.c ...
-en     .././simpleserial/simpleserial.c ...
-en     .././hal/hal.c ...
Compiling:
-en     .././hal//stm32f0_nano/stm32f0_hal_nano.c ...
.
.
Compiling:
Compiling:
-en     .././hal//stm32f0/stm32f0_hal_lowlevel.c ...
.
-en     .././crypto/tiny-AES128-C/aes.c ...
Compiling:
-en     .././crypto/aes-independant.c ...
.
Assembling: .././hal//stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mf
loat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gstabs,-adhlns=objd
ir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././hal/ -I.././hal/
-I.././hal//stm32f0 -I.././hal//stm32f0/CMSIS -I.././hal//stm32f0/CMSIS/co
re -I.././hal//stm32f0/CMSIS/device -I.././hal//stm32f0/Legacy -I.././simp
leserial/ -I.././crypto/ -I.././crypto/tiny-AES128-C .././hal//stm32f0/stm
32f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
.
LINKING:
-en     simpleserial-aes-CWNANO.elf ...
-e Done!
.
.
.
Creating load file for Flash: simpleserial-aes-CWNANO.hex
.
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature s
```

```
impleserial-aes-CWNANO.elf simpleserial-aes-CWNANO.hex
Creating load file for Flash: simpleserial-aes-CWNANO.bin
Creating load file for EEPROM: simpleserial-aes-CWNANO.eep
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature
simpleserial-aes-CWNANO.elf simpleserial-aes-CWNANO.bin
Creating Extended Listing: simpleserial-aes-CWNANO.lss
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load"
\
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-a
es-CWNANO.elf simpleserial-aes-CWNANO.eep || exit 0
arm-none-eabi-objdump -h -S -z simpleserial-aes-CWNANO.elf > simpleserial-
aes-CWNANO.lss
.
Creating Symbol Table: simpleserial-aes-CWNANO.sym
arm-none-eabi-nm -n simpleserial-aes-CWNANO.elf > simpleserial-aes-CWNANO.
sym
Size after:
   text    data     bss     dec     hex filename
   5648     540    1884    8072    1f88 simpleserial-aes-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = TINYAES128C
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
```

In [ ]:
```
cw.program_target(scope, prog, "../../../firmware/mcu/simpleserial-aes/si
```

```
Detected known STMF32: STM32F03xx4/03xx6
Extended erase (0x44), this can take ten seconds or more
Attempting to program 6187 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 6187 bytes
```

Capturing Traces

In [ ]:
```
from tqdm.notebook import trange
import numpy as np
import time

ktp = cw.ktp.Basic()

trace_counts = [1000, 2000, 2500, 5000]
captured_traces = []

for N in trace_counts:
    trace_array = []
    textin_array = []

    key, text = ktp.next()

    target.set_key(key)

    for i in trange(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()

    captured_traces.append([trace_array, textin_array])
```

```
Capturing traces:   0%|          | 0/1000 [00:00<?, ?it/s]
Capturing traces:   0%|          | 0/2000 [00:00<?, ?it/s]
Capturing traces:   0%|          | 0/2500 [00:00<?, ?it/s]
Capturing traces:   0%|          | 0/5000 [00:00<?, ?it/s]
```

Theoretical AES Implementation

In [ ]:

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

def aes_internal(inputdata, key):
    return sbox[inputdata ^ key]
```

DPA Implementation

In [ ]:

```
def calculate_diffs(guess, byteindex, bitnum, textin_array, trace_array):

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

    for trace_index in range(numtraces):
```

```
        hypothetical_leakage = aes_internal(guess, textin_array[trace_ind

        #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)
```

In [ ]:

```
from tqdm.notebook import tnrange
import numpy as np

# Known key for validation
known_key = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
             0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c]

# Store results
key_guesses = [[0]*8 for _ in range(16)]  # best guess per bitnum
bit_rankings = [[0]*8 for _ in range(16)] # rank of correct key for each
best_bits = [0]*16  # best bitnum per subkey

for i in range(len(trace_counts)):
    print(f"\n--- Running DPA with {trace_counts[i]} traces ---")
    trace_array, textin_array = captured_traces[i]

    for subkey in tnrange(0, 4, desc=f"Subkey"):
        best_bit = 0
        best_diff = -1

        for bitnum in range(8):
            max_diffs = [0]*256

            for guess in range(256):
                full_diff_trace = calculate_diffs(guess, subkey, bitnum,
                full_diff_trace = full_diff_trace[(1100 + subkey*60):]
                max_diffs[guess] = np.max(full_diff_trace)

            # Sorted guesses (highest to lowest diff)
            sorted_args = np.argsort(max_diffs)[::-1]
            top_guess = sorted_args[0]
            rank_of_correct = list(sorted_args).index(known_key[subkey])
            correct_diff = max_diffs[known_key[subkey]]

            key_guesses[subkey][bitnum] = top_guess
            bit_rankings[subkey][bitnum] = rank_of_correct

            if correct_diff > best_diff:
                best_diff = correct_diff
                best_bit = bitnum

        best_bits[subkey] = best_bit
        print(f"Subkey {subkey:02d}: Best Bit = {best_bit}, Rank of corre
```

```
--- Running DPA with 1000 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
```

```
Subkey 00: Best Bit = 7, Rank of correct key at that bit = 12
Subkey 01: Best Bit = 4, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 2, Rank of correct key at that bit = 37
Subkey 03: Best Bit = 1, Rank of correct key at that bit = 17

--- Running DPA with 2000 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
Subkey 00: Best Bit = 5, Rank of correct key at that bit = 9
Subkey 01: Best Bit = 4, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 1, Rank of correct key at that bit = 31
Subkey 03: Best Bit = 7, Rank of correct key at that bit = 13

--- Running DPA with 2500 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
Subkey 00: Best Bit = 1, Rank of correct key at that bit = 8
Subkey 01: Best Bit = 4, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 7, Rank of correct key at that bit = 50
Subkey 03: Best Bit = 3, Rank of correct key at that bit = 88

--- Running DPA with 5000 traces ---
Subkey:   0%|          | 0/4 [00:00<?, ?it/s]
Subkey 00: Best Bit = 5, Rank of correct key at that bit = 5
Subkey 01: Best Bit = 3, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 6, Rank of correct key at that bit = 50
Subkey 03: Best Bit = 0, Rank of correct key at that bit = 2
```

In [ ]:

```
trace_array, textin_array = captured_traces[-1]
for subkey in tnrange(0, 16, desc=f"Subkey"):
    best_bit = 0
    best_diff = -1

    for bitnum in range(8):
        max_diffs = [0]*256

        for guess in range(256):
            full_diff_trace = calculate_diffs(guess, subkey, bitnum, text
            full_diff_trace = full_diff_trace[(1100 + subkey*60):]
            max_diffs[guess] = np.max(full_diff_trace)

        # Sorted guesses (highest to lowest diff)
        sorted_args = np.argsort(max_diffs)[::-1]
        top_guess = sorted_args[0]
        rank_of_correct = list(sorted_args).index(known_key[subkey])
        correct_diff = max_diffs[known_key[subkey]]

        key_guesses[subkey][bitnum] = top_guess
        bit_rankings[subkey][bitnum] = rank_of_correct

        if correct_diff > best_diff:
            best_diff = correct_diff
            best_bit = bitnum

    best_bits[subkey] = best_bit
    print(f"Subkey {subkey:02d}: Best Bit = {best_bit}, Rank of correct k
```

```
Subkey:   0%|          | 0/16 [00:00<?, ?it/s]
```

```
Subkey 00: Best Bit = 5, Rank of correct key at that bit = 5
Subkey 01: Best Bit = 3, Rank of correct key at that bit = 0
Subkey 02: Best Bit = 6, Rank of correct key at that bit = 50
Subkey 03: Best Bit = 0, Rank of correct key at that bit = 2
Subkey 04: Best Bit = 1, Rank of correct key at that bit = 57
Subkey 05: Best Bit = 7, Rank of correct key at that bit = 72
Subkey 06: Best Bit = 2, Rank of correct key at that bit = 9
Subkey 07: Best Bit = 4, Rank of correct key at that bit = 19
Subkey 08: Best Bit = 2, Rank of correct key at that bit = 29
Subkey 09: Best Bit = 6, Rank of correct key at that bit = 166
Subkey 10: Best Bit = 5, Rank of correct key at that bit = 12
Subkey 11: Best Bit = 4, Rank of correct key at that bit = 84
Subkey 12: Best Bit = 6, Rank of correct key at that bit = 14
Subkey 13: Best Bit = 1, Rank of correct key at that bit = 20
Subkey 14: Best Bit = 2, Rank of correct key at that bit = 23
Subkey 15: Best Bit = 6, Rank of correct key at that bit = 7
```

Questions from Lab 3

What about using other bits in the output as the rule to partition power traces, e.g., bit-X?

Using other bits (bit 0 to bit 7) in the intermediate value (e.g., S-box output) to partition traces is a valid and often useful strategy. This generalizes the traditional DPA approach, which often focuses only on bit-0. Each bit of the S-box output leaks differently based on circuit design, operation timing, and power measurement noise. Some bits may leak more reliably depending on how the microcontroller processes those bits internally.

For the recovery of each subkey (byte, or round key), what would be the best bit-X, e.g., yielding the most significant differences compared to other guesses?

The above output shows that the best bit is different for each subkey. I ran for only 4 subkeys because it was taking a huge amount of time for 16.

Does it exist the best bit-X that is simply the best for each subkey recovery? No, there is usually not a single global best bit for all subkeys which is clearly evident from the above results. This is due to how AES processes data and how leakage manifests across instructions and cycles. You can determine this by analyzing the distribution of `best_bits[subkey]`:

- If one bit dominates (e.g., bit 0 is best for most subkeys), then it's a good default.
- If the best bit is spread out evenly, per-subkey selection is better.

If bit-0 is not the best global partition policy, how would you choose your partition policy?

We can adaptively select the bit per subkey by running DPA on all bits for every subkey. Then measure the difference peak. Select the bit-X with the highest leakage as the partition rule for that subkey.

Explore all the questions above with the number of traces in 1K, 2K, 2.5K (default), and 5K.

Based on the results of running DPA with varying numbers of traces, we observed the following trends:

- As the number of traces increases, the DPA attack becomes more effective. The correct key bytes are more likely to be ranked at the top.
- The ranking of the correct key byte improves as the trace count (N) increases. Lower rank values indicate higher confidence in the guessed key byte.
- While Bit-0 is commonly used in DPA, it is not always the optimal choice. At lower trace counts, certain bits—such as Bit-0 or Bit-7—may exhibit stronger leakage, but this varies between subkeys.
- **At Low Trace Counts (e.g., 1,000 traces):**
  - Greater variability is observed in which bit provides the best results per subkey.
  - The correct key may not rank highly, leading to inaccurate or failed key recovery for some subkeys.
- **At Higher Trace Counts (e.g., 5,000 traces):**
  - The choice of best-leaking bit for each subkey becomes more consistent.
  - The correct key is more reliably identified, with improved ranking across all subkeys.

Summary Table

Traces	Subkey	Best Bit	Rank of Correct Key
1000	00	7	12
1000	01	4	0
1000	02	2	37
1000	03	1	17
2000	00	5	9
2000	01	4	0
2000	02	1	31
2000	03	7	13
2500	00	1	8
2500	01	4	0
2500	02	7	50
2500	03	3	88
5000	00	5	5
5000	01	3	0
5000	02	6	50
5000	03	0	2

Appendix

Part 3, Topic 3: DPA on Firmware Implementation of AES

NOTE: This lab references some (commercial) training material on ChipWhisperer.io. You can freely execute and use the lab per the open-source license (including using it in your own courses if you distribute similarly), but you must maintain notice about this source location. Consider joining our training course to enjoy the full experience.

SUMMARY: *In the previous lab, you saw how a single bit of information can be used to recover an entire byte of the AES key. Remember, this works due to the S-Box being present in the data flow that we are attacking.*

Next, we'll see how to use power analysis instead of an actual bit value. With this technique, the goal is to separate the traces by a bit in the result of the SBox output (it doesn't matter which one): if that bit is 1, its group of traces should, on average, have higher power consumption during the SBox operation than the other set.

This is all based on the assumption we discussed in the slides and saw in earlier labs: there is some consistent relationship between the value of bits on the data bus and the power consumption in the device.

LEARNING OUTCOMES:

- Using a power measurement to 'validate' a possible device model.
- Detecting the value of a single bit using power measurement.
- Breaking AES using the classic DPA attack.

Prerequisites

Hold up! Before you continue, check you've done the following tutorials:

- ☑ Jupyter Notebook Intro (you should be OK with plotting & running blocks).
- ☑ SCA101 Intro (you should have an idea of how to get hardware-specific versions running).
- ☑ Breaking AES Using a Single Bit (we'll build on your previous work).

In [ ]:
```
SCOPETYPE = 'CWNANO'
PLATFORM = 'CWNANO'
CRYPTO_TARGET = 'TINYAES128C'
VERSION = 'HARDWARE'
```

In [ ]:
```
if VERSION == 'HARDWARE':
    %run "Lab 3_3 - DPA on Firmware Implementation of AES (HARDWARE).ipyn
elif VERSION == 'SIMULATED':
    %run "Lab 3_3 - DPA on Firmware Implementation of AES (SIMULATED).ipy
```

```
INFO: Caught exception on reconnecting to target - attempting to reconnect
to scope first.
INFO: This is a work-around when USB has died without Python knowing. Igno
re errors above this line.
INFO: Found ChipWhisperer😍
Building for platform CWNANO with CRYPTO_TARGET=TINYAES128C
SS_VER set to SS_VER_1_1
SS_VER set to SS_VER_1_1
Blank crypto options, building for AES128
arm-none-eabi-gcc (15:10.3-2021.07-4) 10.3.1 20210621 (release)
Copyright (C) 2020 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOS
E.

mkdir -p objdir-CWNANO
.
Welcome to another exciting ChipWhisperer target build!!
.
.
.
.
Compiling:
Compiling:
Compiling:
Compiling:
-en     simpleserial-aes.c ...
-en     .././simpleserial/simpleserial.c ...
.
-en     .././hal/hal.c ...
-en     .././hal//stm32f0_nano/stm32f0_hal_nano.c ...
Compiling:
-en     .././hal//stm32f0/stm32f0_hal_lowlevel.c ...
.
.
Compiling:
Compiling:
.
-en     .././crypto/tiny-AES128-C/aes.c ...
-en     .././crypto/aes-independant.c ...
Assembling: .././hal//stm32f0/stm32f0_startup.S
arm-none-eabi-gcc -c -mcpu=cortex-m0 -I. -x assembler-with-cpp -mthumb -mf
loat-abi=soft -ffunction-sections -DF_CPU=7372800 -Wa,-gstabs,-adhlns=objd
ir-CWNANO/stm32f0_startup.lst -I.././simpleserial/ -I.././hal/ -I.././hal/
-I.././hal//stm32f0 -I.././hal//stm32f0/CMSIS -I.././hal//stm32f0/CMSIS/co
re -I.././hal//stm32f0/CMSIS/device -I.././hal//stm32f0/Legacy -I.././simp
leserial/ -I.././crypto/ -I.././crypto/tiny-AES128-C .././hal//stm32f0/stm
32f0_startup.S -o objdir-CWNANO/stm32f0_startup.o
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
-e Done!
.
LINKING:
-en     simpleserial-aes-CWNANO.elf ...
-e Done!
.
```

```
.
.
Creating load file for Flash: simpleserial-aes-CWNANO.hex
arm-none-eabi-objcopy -O ihex -R .eeprom -R .fuse -R .lock -R .signature s
impleserial-aes-CWNANO.elf simpleserial-aes-CWNANO.hex
.
Creating load file for Flash: simpleserial-aes-CWNANO.bin
Creating load file for EEPROM: simpleserial-aes-CWNANO.eep
arm-none-eabi-objcopy -O binary -R .eeprom -R .fuse -R .lock -R .signature
simpleserial-aes-CWNANO.elf simpleserial-aes-CWNANO.bin
arm-none-eabi-objcopy -j .eeprom --set-section-flags=.eeprom="alloc,load"
\
--change-section-lma .eeprom=0 --no-change-warnings -O ihex simpleserial-a
es-CWNANO.elf simpleserial-aes-CWNANO.eep || exit 0
Creating Extended Listing: simpleserial-aes-CWNANO.lss
arm-none-eabi-objdump -h -S -z simpleserial-aes-CWNANO.elf > simpleserial-
aes-CWNANO.lss
.
Creating Symbol Table: simpleserial-aes-CWNANO.sym
arm-none-eabi-nm -n simpleserial-aes-CWNANO.elf > simpleserial-aes-CWNANO.
sym
Size after:
   text    data     bss     dec     hex filename
   5648     540    1884    8072    1f88 simpleserial-aes-CWNANO.elf
+--------------------------------------------------------
+ Default target does full rebuild each time.
+ Specify buildtarget == allquick == to avoid full rebuild
+--------------------------------------------------------
+--------------------------------------------------------
+ Built for platform CWNANO Built-in Target (STM32F030) with:
+ CRYPTO_TARGET = TINYAES128C
+ CRYPTO_OPTIONS = AES128C
+--------------------------------------------------------
Detected known STMF32: STM32F03xx4/03xx6
Extended erase (0x44), this can take ten seconds or more
Attempting to program 6187 bytes at 0x8000000
STM32F Programming flash...
STM32F Reading flash...
Verified flash OK, 6187 bytes
Capturing traces:   0%|          | 0/2500 [00:00<?, ?it/s]
```

AES Model

No need to remember the complex model from before - we can instead just jump right into the AES model! Copy your AES model you developed in the previous lab below & run it:

In [ ]:

```
# ###################
# Add your code here
# ###################
#raise NotImplementedError("Add your code here, and delete this.")

# ###################
# START SOLUTION
# ###################
sbox = [
    # 0    1    2    3    4    5    6    7    8    9    a    b    c    d
    0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7
```

```
    0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4
    0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8
    0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27
    0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3
    0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c
    0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c
    0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff
    0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d
    0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e
    0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95
    0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a
    0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd
    0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1
    0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55
    0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54
]

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

You can verify the model works by running the following blocks, just like last time:

In [ ]:
```
#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!")
```

```
✔️ OK to continue!
```

AES Power Watcher

The next step is to send random data to the device, and observe the power consumption during the encryption.

The idea is that we will use a capture loop like this:

```
print(scope)
for i in trange(N, desc='Capturing traces'):
    key, text = ktp.next()  # manual creation of a key,
text pair can be substituted here

    trace = cw.capture_trace(scope, target, text, key)
    if trace is None:
        continue
    traces.append(trace)
    plot.send(trace)

#Convert traces to numpy arrays
trace_array = np.asarray([trace.wave for trace in traces])
textin_array = np.asarray([trace.textin for trace in
traces])
known_keys = np.asarray([trace.key for trace in traces])
# for fixed key, these keys are all the same
```

Depending what you are using, you can complete this either by:

- Capturing new traces from a physical device.
- Reading pre-recorded data from a file.

You get to choose your adventure - see the two notebooks with the same name of this, but called `(SIMULATED)` or `(HARDWARE)` to continue. Inside those notebooks you should get some code to copy into the following section, which will define the capture function.

Be sure you get the `"✔️ OK to continue!"` print once you run the next cell, otherwise things will fail later on!

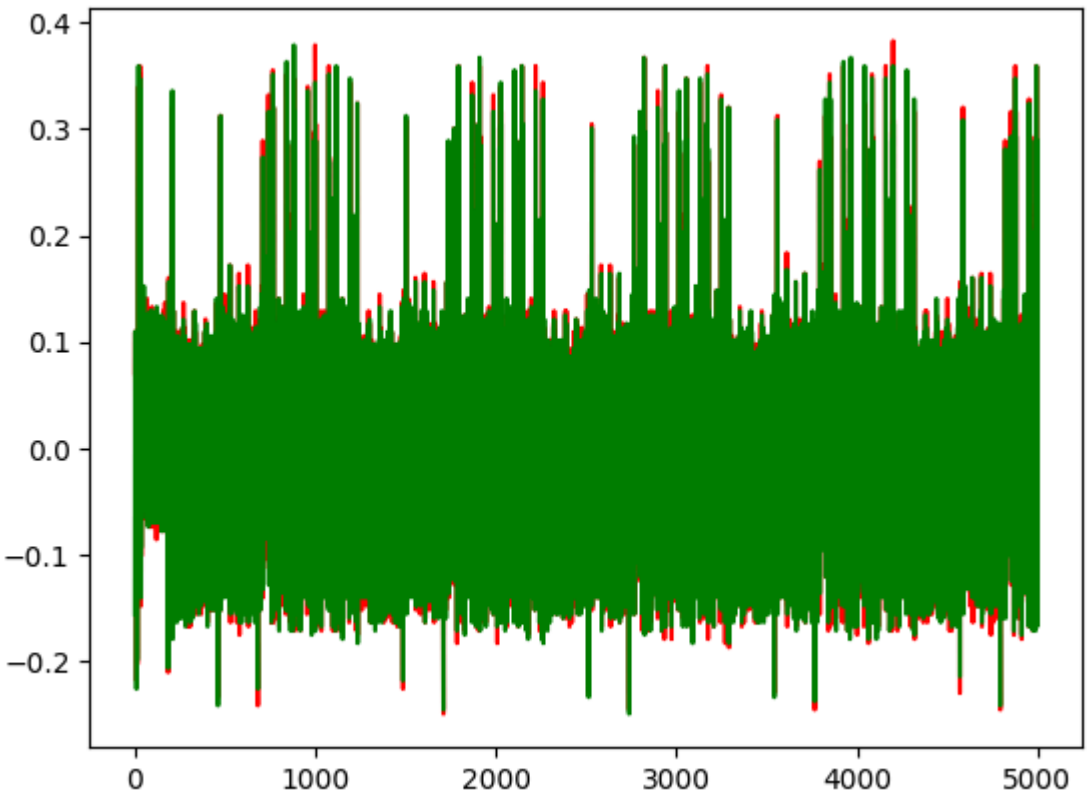
In [ ]:
```
#
# Perform the capture, resulting in trace_array and textin_array of 2500
#
# raise NotImplementedError("Add your code here, and delete this.")

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

```
✔️ OK to continue!
```

What's this data look like? Try plotting a trace or two here:

In [ ]:
```
# raise NotImplementedError("Add your code here, and delete this.")
%matplotlib inline
import matplotlib.pylab as plt
plt.figure()
plt.plot(trace_array[0], 'r')
plt.plot(trace_array[1], 'g')
plt.show()
```



OK interesting - so we've got data! And what about the format of the input data?

In [ ]:
```
print(textin_array[0])
print(textin_array[1])
```

```
CWbytearray(b'07 7c a5 e0 27 7c 43 4a c9 ab 94 c4 6d f3 9b 30')
CWbytearray(b'52 e1 d3 85 85 e0 89 05 ee d3 68 c8 5b 47 b1 06')
```

AES Guesser - One Byte

The attack now needs a way of splitting traces into two groups, depending on the state of a bit in our "guessed" value. We're going to start easy by guessing a single byte of the AES key at a time.

To start with - define the number of traces & number of points in each trace. You can use the following example code, just run this block:

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

If you remember from the slides - our algorithm looks like this:

```
for key_byte_guess_value in [0, 1, 2, 3, ... 253, 254,
255]:

    one_list = empty list
    zero_list = empty list

    for trace_index in [0, 1, 2, 3, ..., numtraces]:

        input_byte = textin_array[trace_index]
[byte_to_attack]

        #Get a hypothetical leakage list - use
aes_internal(guess, input_byte)

        if hypothetical_leakage bit 0 is 1:
            append trace_array[trace_index] to one_list
        else:
            append trace_array[trace_index] to zero_list

    one_avg = average of one_list
    zero_avg = average of zero_list

    max_diff_value = maximum of ABS(one_avg - zero_avg)
```

To get the average of your `one_list` and `zero_list` you can use numpy:

```
import numpy as np
avg_one_list = np.asarray(one_list).mean(axis=0)
```

The important thing here is the `axis=0`, which does an average so the resulting array is done across all traces (not just the average value of one trace, but the average of each point index *across all traces*).

To help you do some testing - let me tell you that the correct value of byte 0 is `0x2B`. You can use this to validate that your solution is working on the first byte. If you get stuck - see some hints below (but give it a try first).

What you should see is an output of the maximum value between the two average groups be higher for the `0x2B` value. For example, priting the maximum SAD value from an example loop looks like this for me:

```
Guessing 28: 0.001397
Guessing 29: 0.000927
Guessing 2a: 0.001953
Guessing 2b: 0.005278
Guessing 2c: 0.000919
Guessing 2d: 0.002510
Guessing 2e: 0.001241
Guessing 2f: 0.001242
```

Note the value of `0.005278` for `0x2B` - this is higher than the others which range from `0.000927` to `0.002510`.

In [ ]:
```
import numpy as np
mean_diffs = np.zeros(256)

guessed_byte = 0

for guess in range(0, 256):

    one_list = []
    zero_list = []

    for trace_index in range(numtraces):

        #Get a hypothetical leakage list - use aes_internal(guess, input_
        hypothetical_leakage = aes_internal(guess, textin_array[trace_ind

        #Mask off the lowest bit - is it 0 or 1? Depending on that add tr
        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[guess] = np.max(abs(one_avg - zero_avg))

    print("Guessing %02x: %f"%(guess, mean_diffs[guess]))
```

```
Guessing 00: 0.001572
Guessing 01: 0.004507
Guessing 02: 0.002538
Guessing 03: 0.001830
Guessing 04: 0.001809
Guessing 05: 0.001601
Guessing 06: 0.001444
Guessing 07: 0.001679
Guessing 08: 0.001871
Guessing 09: 0.007052
Guessing 0a: 0.003506
Guessing 0b: 0.001433
Guessing 0c: 0.004204
Guessing 0d: 0.001284
Guessing 0e: 0.001425
Guessing 0f: 0.001906
Guessing 10: 0.002533
Guessing 11: 0.001842
Guessing 12: 0.004884
Guessing 13: 0.001944
Guessing 14: 0.003367
Guessing 15: 0.002770
Guessing 16: 0.002931
Guessing 17: 0.002819
Guessing 18: 0.001832
Guessing 19: 0.004052
Guessing 1a: 0.002036
Guessing 1b: 0.002908
Guessing 1c: 0.002227
Guessing 1d: 0.001575
Guessing 1e: 0.003222
Guessing 1f: 0.002235
Guessing 20: 0.002359
Guessing 21: 0.002094
Guessing 22: 0.001735
Guessing 23: 0.001613
Guessing 24: 0.001663
Guessing 25: 0.002431
Guessing 26: 0.002538
Guessing 27: 0.002836
Guessing 28: 0.001514
Guessing 29: 0.003929
Guessing 2a: 0.004077
Guessing 2b: 0.006283
Guessing 2c: 0.001542
Guessing 2d: 0.001532
Guessing 2e: 0.001515
Guessing 2f: 0.002437
Guessing 30: 0.001443
Guessing 31: 0.002546
Guessing 32: 0.003575
Guessing 33: 0.003013
Guessing 34: 0.002781
Guessing 35: 0.005186
Guessing 36: 0.003841
Guessing 37: 0.002829
Guessing 38: 0.004214
Guessing 39: 0.003811
Guessing 3a: 0.001289
Guessing 3b: 0.009505
```

```
Guessing 3c: 0.003942
Guessing 3d: 0.006438
Guessing 3e: 0.001597
Guessing 3f: 0.005984
Guessing 40: 0.003183
Guessing 41: 0.002011
Guessing 42: 0.002222
Guessing 43: 0.004427
Guessing 44: 0.002570
Guessing 45: 0.001410
Guessing 46: 0.002310
Guessing 47: 0.001364
Guessing 48: 0.003714
Guessing 49: 0.002805
Guessing 4a: 0.002116
Guessing 4b: 0.003322
Guessing 4c: 0.001767
Guessing 4d: 0.002017
Guessing 4e: 0.002009
Guessing 4f: 0.002274
Guessing 50: 0.001893
Guessing 51: 0.001346
Guessing 52: 0.001537
Guessing 53: 0.003074
Guessing 54: 0.001641
Guessing 55: 0.001581
Guessing 56: 0.002142
Guessing 57: 0.001994
Guessing 58: 0.005742
Guessing 59: 0.002538
Guessing 5a: 0.001332
Guessing 5b: 0.001547
Guessing 5c: 0.001468
Guessing 5d: 0.001420
Guessing 5e: 0.001286
Guessing 5f: 0.003255
Guessing 60: 0.001449
Guessing 61: 0.003024
Guessing 62: 0.002807
Guessing 63: 0.003857
Guessing 64: 0.006102
Guessing 65: 0.001626
Guessing 66: 0.001436
Guessing 67: 0.004094
Guessing 68: 0.002339
Guessing 69: 0.003094
Guessing 6a: 0.001585
Guessing 6b: 0.004942
Guessing 6c: 0.002090
Guessing 6d: 0.001634
Guessing 6e: 0.002773
Guessing 6f: 0.003114
Guessing 70: 0.002384
Guessing 71: 0.001574
Guessing 72: 0.003250
Guessing 73: 0.002086
Guessing 74: 0.002553
Guessing 75: 0.001545
Guessing 76: 0.001878
Guessing 77: 0.002193
```

```
Guessing 78: 0.003499
Guessing 79: 0.002551
Guessing 7a: 0.001380
Guessing 7b: 0.004881
Guessing 7c: 0.006763
Guessing 7d: 0.001606
Guessing 7e: 0.004570
Guessing 7f: 0.004058
Guessing 80: 0.001644
Guessing 81: 0.002480
Guessing 82: 0.001871
Guessing 83: 0.002208
Guessing 84: 0.006622
Guessing 85: 0.001595
Guessing 86: 0.001606
Guessing 87: 0.003306
Guessing 88: 0.003017
Guessing 89: 0.002517
Guessing 8a: 0.002510
Guessing 8b: 0.002075
Guessing 8c: 0.003473
Guessing 8d: 0.002823
Guessing 8e: 0.001773
Guessing 8f: 0.004629
Guessing 90: 0.002682
Guessing 91: 0.001731
Guessing 92: 0.003216
Guessing 93: 0.001333
Guessing 94: 0.003603
Guessing 95: 0.002472
Guessing 96: 0.003030
Guessing 97: 0.002884
Guessing 98: 0.002545
Guessing 99: 0.004067
Guessing 9a: 0.002137
Guessing 9b: 0.001702
Guessing 9c: 0.001874
Guessing 9d: 0.003103
Guessing 9e: 0.003332
Guessing 9f: 0.001692
Guessing a0: 0.002712
Guessing a1: 0.001418
Guessing a2: 0.001839
Guessing a3: 0.002494
Guessing a4: 0.001963
Guessing a5: 0.006723
Guessing a6: 0.002213
Guessing a7: 0.001566
Guessing a8: 0.004733
Guessing a9: 0.002857
Guessing aa: 0.001806
Guessing ab: 0.003324
Guessing ac: 0.001404
Guessing ad: 0.003789
Guessing ae: 0.003330
Guessing af: 0.001924
Guessing b0: 0.004469
Guessing b1: 0.003514
Guessing b2: 0.002720
Guessing b3: 0.001910
```

```
Guessing b4: 0.002563
Guessing b5: 0.002187
Guessing b6: 0.002977
Guessing b7: 0.002326
Guessing b8: 0.001657
Guessing b9: 0.002624
Guessing ba: 0.001573
Guessing bb: 0.001584
Guessing bc: 0.002905
Guessing bd: 0.002117
Guessing be: 0.001972
Guessing bf: 0.002534
Guessing c0: 0.001959
Guessing c1: 0.001247
Guessing c2: 0.002064
Guessing c3: 0.002335
Guessing c4: 0.002429
Guessing c5: 0.001365
Guessing c6: 0.001531
Guessing c7: 0.003317
Guessing c8: 0.001855
Guessing c9: 0.001971
Guessing ca: 0.001546
Guessing cb: 0.002500
Guessing cc: 0.001709
Guessing cd: 0.002340
Guessing ce: 0.002342
Guessing cf: 0.001516
Guessing d0: 0.002081
Guessing d1: 0.002394
Guessing d2: 0.002615
Guessing d3: 0.004402
Guessing d4: 0.002980
Guessing d5: 0.002840
Guessing d6: 0.002691
Guessing d7: 0.004781
Guessing d8: 0.001541
Guessing d9: 0.004554
Guessing da: 0.002224
Guessing db: 0.004475
Guessing dc: 0.004072
Guessing dd: 0.003056
Guessing de: 0.002696
Guessing df: 0.002733
Guessing e0: 0.001826
Guessing e1: 0.002055
Guessing e2: 0.002259
Guessing e3: 0.002337
Guessing e4: 0.002920
Guessing e5: 0.002122
Guessing e6: 0.002366
Guessing e7: 0.001670
Guessing e8: 0.001576
Guessing e9: 0.002451
Guessing ea: 0.001717
Guessing eb: 0.002660
Guessing ec: 0.002516
Guessing ed: 0.002701
Guessing ee: 0.001533
Guessing ef: 0.005719
```

```
Guessing f0: 0.001485
Guessing f1: 0.001533
Guessing f2: 0.001385
Guessing f3: 0.001862
Guessing f4: 0.002169
Guessing f5: 0.002951
Guessing f6: 0.004332
Guessing f7: 0.004018
Guessing f8: 0.005157
Guessing f9: 0.001824
Guessing fa: 0.001460
Guessing fb: 0.001689
Guessing fc: 0.004892
Guessing fd: 0.003406
Guessing fe: 0.001499
Guessing ff: 0.003385
```

Hint 1: General Program Flow

You can use the following general program flow to help you implement the outer loop above:

In [ ]:
```python
#Hint #1 - General Program Flow
import numpy as np
mean_diffs = np.zeros(256)

guessed_byte = 0

for guess in range(0, 256):

    one_list = []
    zero_list = []

    for trace_index in range(numtraces):
        #Inside here do the steps shown above
        pass

    #Do extra steps to average one_list and zero_list
```

Hint 2: Example of Two Different Key Guesses

We aren't fully going to give it away (see `SOLN` notebook if you want that), but here is how you can generate two differences, for `0x2B` and `0xFF`. If you're totally stuck you can use the following code to base what should be inside the loops on.

In [ ]:
```python
import numpy as np
mean_diffs = np.zeros(256)

### Code to do guess of byte 0 set to 0x2B
guessed_byte = 0
guess = 0x2B

one_list = []
zero_list = []

for trace_index in range(numtraces):
```

```
    hypothetical_leakage = aes_internal(guess, textin_array[trace_index][

    #Mask off the lowest bit - is it 0 or 1? Depending on that add trace
    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))

print("Max SAD for 0x2B: {:1}".format(mean_diffs_2b))

### Code to do guess of byte 0 set to 0xFF
guessed_byte = 0
guess = 0xFF

one_list = []
zero_list = []

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

    #Mask off the lowest bit - is it 0 or 1? Depending on that add trace
    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_ff = np.max(abs(one_avg - zero_avg))

print("Max SAD for 0xFF: {:1}".format(mean_diffs_ff))
```

```
Max SAD for 0x2B: 0.0062833446065526655
Max SAD for 0xFF: 0.0033847127405046695
```

Ranking Guesses

You'll also want to rank some of your guesses (we assume). This will help you identify the most likely value. The best way to do this is build a list of the maximum difference values for each key:

```
mean_diffs = [0]*256

for key_byte_guess_value in [0, 1, 2, 3, ... 253, 254,
255]:

    *** CODE FROM BEFORE***
    max_diff_value = maximum of ABS(one_avg - zero_avg)
    mean_diffs[key_byte_guess_value] = max_diff_value
```

If you modify your previous code, it will generate a list of maximum differences in a list. This list will look like:

```
[0.002921, 0.001923, 0.005131, ..., 0.000984]
```

Where the *index* of the list is the value of the key guess. We can use `np.argsort` which generates a new list showing the *indicies* that would sort an original list (you should have learned about `argsort` in the previous lab too):

So for example, run the following to see it in action on the list `[1.0, 0.2, 3.4, 0.01]`:

In [ ]:
```
np.argsort([1.0, 0.2, 3.4, 0.01])
```

Out[ ]:
```
array([3, 1, 0, 2])
```

This should return `[3, 1, 0, 2]` - that is the order of lowest to highest. To change from highest to lowest, remember you just add `[::-1]` at the end of it like `np.argsort([1.0, 0.2, 3.4, 0.01])[::-1]`.

Try using the `np.argsort` function to output the most likely key values from your attack.

Plotting Differences

Before we move on - you should take a look at various plots of these differences. They will play in something called the *ghost peak* problem.

We're going to now define a function called `calculate_diffs()` that implements our attacks (you can replace this with your own function or keep this one for now):

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

    one_list = []
    zero_list = []

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

        #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)
```

Try plotting the difference between various bytes. For byte 0, remember `0x2B` is the correct value. Zoom in on the plots and see how the correct key should have a much larger difference.

Sometimes we get *ghost peaks* which are incorrect peaks. So far we're assuming there is a single "best" solution for the key - we may need to get fancy and put a threshold whereby we have several candidates for the correct key. For now let's just plot a handful of examples:

In [ ]:
```
cw.plot(calculate_diffs(0x2B)) * cw.plot(calculate_diffs(0x2C)) * cw.plot
```

Out[ ]:

Here is what it should look like:

You'll notice when we rank the bytes we just use the maximum value of any peak. There's lots more you could learn from these graphs, such as the location of the peak, or if there are multiple peaks in the graph. But for now we're just going to keep with the

AES Guesser - All Bytes

Alright - good job! You've got a single byte and some DPA plots up. Now let's move onward and guess *all* of the bytes.

Doing this requires a little more effort than before. Taking your existing guessing function, you're going to wrap a larger loop around the outside of it like this:

```
for subkey in range(0,16):
    #Rest of code from before!
```

In [ ]:
```
from tqdm.notebook import trange
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,

for subkey in trange(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)
        max_diffs[guess] = np.max(full_diff_trace)
        full_diffs[guess] = full_diff_trace

    #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_gues

    #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]))

    print("\n")
```

```
Attacking Subkey:   0%|          | 0/16 [00:00<?, ?it/s]
```

```
Subkey  0 - most likely 67 (actual 2B)
 Top 5 guesses:
   67 - Diff = 0.006610
   2B - Diff = 0.006290
   09 - Diff = 0.005949
   06 - Diff = 0.005807
   4D - Diff = 0.005537


Subkey  1 - most likely 9B (actual 7E)
 Top 5 guesses:
   9B - Diff = 0.006060
   AB - Diff = 0.005541
   96 - Diff = 0.005286
   3C - Diff = 0.005222
   74 - Diff = 0.005081


Subkey  2 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.006528
   5C - Diff = 0.005530
   B8 - Diff = 0.005272
   D3 - Diff = 0.005191
   C0 - Diff = 0.005077


Subkey  3 - most likely 52 (actual 16)
 Top 5 guesses:
   52 - Diff = 0.007875
   0B - Diff = 0.007829
   16 - Diff = 0.006548
   D1 - Diff = 0.006106
   A4 - Diff = 0.006043


Subkey  4 - most likely 49 (actual 28)
 Top 5 guesses:
   49 - Diff = 0.007173
   28 - Diff = 0.006761
   6C - Diff = 0.005939
   61 - Diff = 0.005384
   ED - Diff = 0.005312


Subkey  5 - most likely 73 (actual AE)
 Top 5 guesses:
   73 - Diff = 0.007094
   AE - Diff = 0.006200
   A5 - Diff = 0.006185
   BA - Diff = 0.006017
   1F - Diff = 0.005923


Subkey  6 - most likely D2 (actual D2)
 Top 5 guesses:
   D2 - Diff = 0.006811
   93 - Diff = 0.006074
   B6 - Diff = 0.005806
   12 - Diff = 0.005723
```

```
   70 - Diff = 0.005685


Subkey  7 - most likely 30 (actual A6)
 Top 5 guesses:
   30 - Diff = 0.006136
   14 - Diff = 0.005966
   90 - Diff = 0.005950
   77 - Diff = 0.005676
   7D - Diff = 0.005672


Subkey  8 - most likely AB (actual AB)
 Top 5 guesses:
   AB - Diff = 0.005828
   77 - Diff = 0.005724
   F0 - Diff = 0.005594
   67 - Diff = 0.005360
   B4 - Diff = 0.005099


Subkey  9 - most likely F7 (actual F7)
 Top 5 guesses:
   F7 - Diff = 0.005889
   51 - Diff = 0.005173
   C4 - Diff = 0.005091
   0D - Diff = 0.004821
   41 - Diff = 0.004818


Subkey 10 - most likely 16 (actual 15)
 Top 5 guesses:
   16 - Diff = 0.007881
   7B - Diff = 0.007756
   45 - Diff = 0.007000
   F0 - Diff = 0.006323
   15 - Diff = 0.006109


Subkey 11 - most likely F8 (actual 88)
 Top 5 guesses:
   F8 - Diff = 0.007431
   88 - Diff = 0.006527
   AB - Diff = 0.006402
   8A - Diff = 0.006398
   A7 - Diff = 0.005581


Subkey 12 - most likely BC (actual 09)
 Top 5 guesses:
   BC - Diff = 0.006620
   13 - Diff = 0.005459
   09 - Diff = 0.005410
   35 - Diff = 0.005366
   41 - Diff = 0.005346


Subkey 13 - most likely D3 (actual CF)
 Top 5 guesses:
   D3 - Diff = 0.006179
```

```
  CB - Diff = 0.005933
  C4 - Diff = 0.005719
  CF - Diff = 0.005707
  E3 - Diff = 0.005242


Subkey 14 - most likely 29 (actual 4F)
 Top 5 guesses:
  29 - Diff = 0.006403
  4F - Diff = 0.006189
  11 - Diff = 0.006081
  7C - Diff = 0.005913
  6C - Diff = 0.005697


Subkey 15 - most likely 96 (actual 3C)
 Top 5 guesses:
  96 - Diff = 0.006832
  A2 - Diff = 0.006397
  8A - Diff = 0.006342
  FB - Diff = 0.006278
  1A - Diff = 0.006218
```

🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳🥳 Congrats - you did it!!!!

Hopefully the above worked - but we're going to go a little further to understand how to apply this in case it didn't work right away (or it almost worked).

Ghost Peaks

Maybe the previous didn't actually recover the full key? No need to worry - there are a few reasons for this. One artifact of a DPA attack is you get another strong peak that isn't the correct key (which can be a ghost peak).

We're going to get into more efficient attacks later, but for now, let's look at some solutions:

- Increase the number of traces recorded.
- Change the targetted bit (& combine solutions from multiple bits).
- Window the input data.

The first one is the brute-force option: go from 2500 to 5000 or even 10000 power traces. As you add more data, you may find the problem is reduced. But real ghost peaks may not disappear, so we need to move onto other solutions.

Before we begin - we're going to give you a "known good" DPA attack script we're going to build on. This uses the `calculate_diffs()` function defined earlier.

Run the following block (will take a bit of time):

In [ ]:
```
from tqdm.notebook import trange
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,

#Which bit to target
bitnum = 0

full_diffs_list = []

for subkey in trange(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)
        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_gues

    #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]))

    print("\n")
```

```
Attacking Subkey:   0%|          | 0/16 [00:00<?, ?it/s]
```

```
Subkey  0 - most likely 67 (actual 2B)
 Top 5 guesses:
   67 - Diff = 0.006610
   2B - Diff = 0.006290
   09 - Diff = 0.005949
   06 - Diff = 0.005807
   4D - Diff = 0.005537


Subkey  1 - most likely 9B (actual 7E)
 Top 5 guesses:
   9B - Diff = 0.006060
   AB - Diff = 0.005541
   96 - Diff = 0.005286
   3C - Diff = 0.005222
   74 - Diff = 0.005081


Subkey  2 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.006528
   5C - Diff = 0.005530
   B8 - Diff = 0.005272
   D3 - Diff = 0.005191
   C0 - Diff = 0.005077


Subkey  3 - most likely 52 (actual 16)
 Top 5 guesses:
   52 - Diff = 0.007875
   0B - Diff = 0.007829
   16 - Diff = 0.006548
   D1 - Diff = 0.006106
   A4 - Diff = 0.006043


Subkey  4 - most likely 49 (actual 28)
 Top 5 guesses:
   49 - Diff = 0.007173
   28 - Diff = 0.006761
   6C - Diff = 0.005939
   61 - Diff = 0.005384
   ED - Diff = 0.005312


Subkey  5 - most likely 73 (actual AE)
 Top 5 guesses:
   73 - Diff = 0.007094
   AE - Diff = 0.006200
   A5 - Diff = 0.006185
   BA - Diff = 0.006017
   1F - Diff = 0.005923


Subkey  6 - most likely D2 (actual D2)
 Top 5 guesses:
   D2 - Diff = 0.006811
   93 - Diff = 0.006074
   B6 - Diff = 0.005806
   12 - Diff = 0.005723
```

```
   70 - Diff = 0.005685


Subkey  7 - most likely 30 (actual A6)
 Top 5 guesses:
   30 - Diff = 0.006136
   14 - Diff = 0.005966
   90 - Diff = 0.005950
   77 - Diff = 0.005676
   7D - Diff = 0.005672


Subkey  8 - most likely AB (actual AB)
 Top 5 guesses:
   AB - Diff = 0.005828
   77 - Diff = 0.005724
   F0 - Diff = 0.005594
   67 - Diff = 0.005360
   B4 - Diff = 0.005099


Subkey  9 - most likely F7 (actual F7)
 Top 5 guesses:
   F7 - Diff = 0.005889
   51 - Diff = 0.005173
   C4 - Diff = 0.005091
   0D - Diff = 0.004821
   41 - Diff = 0.004818


Subkey 10 - most likely 16 (actual 15)
 Top 5 guesses:
   16 - Diff = 0.007881
   7B - Diff = 0.007756
   45 - Diff = 0.007000
   F0 - Diff = 0.006323
   15 - Diff = 0.006109


Subkey 11 - most likely F8 (actual 88)
 Top 5 guesses:
   F8 - Diff = 0.007431
   88 - Diff = 0.006527
   AB - Diff = 0.006402
   8A - Diff = 0.006398
   A7 - Diff = 0.005581


Subkey 12 - most likely BC (actual 09)
 Top 5 guesses:
   BC - Diff = 0.006620
   13 - Diff = 0.005459
   09 - Diff = 0.005410
   35 - Diff = 0.005366
   41 - Diff = 0.005346


Subkey 13 - most likely D3 (actual CF)
 Top 5 guesses:
   D3 - Diff = 0.006179
```

```
  CB - Diff = 0.005933
  C4 - Diff = 0.005719
  CF - Diff = 0.005707
  E3 - Diff = 0.005242


Subkey 14 - most likely 29 (actual 4F)
 Top 5 guesses:
  29 - Diff = 0.006403
  4F - Diff = 0.006189
  11 - Diff = 0.006081
  7C - Diff = 0.005913
  6C - Diff = 0.005697


Subkey 15 - most likely 96 (actual 3C)
 Top 5 guesses:
  96 - Diff = 0.006832
  A2 - Diff = 0.006397
  8A - Diff = 0.006342
  FB - Diff = 0.006278
  1A - Diff = 0.006218
```

This block should now print some *next top guesses* - in this case just the next top 5 guesses, but you can extend this if you wish. It's also keeping a copy of all the *difference* traces (unlike before where it threw them away).

Plotting Peaks

After it runs, select a subkey that is either wrong or has very close "next best guesses". For example, the following shows the output for Subkey 5 is actually wrong - the correct guess (`0xAE`) has been ranked as option 5.

```
Subkey  5 - most likely CB (actual AE)
 Top 5 guesses:
   CB - Diff = 0.003006
   C5 - Diff = 0.002984
   AE - Diff = 0.002739
   3C - Diff = 0.002674
   2F - Diff = 0.002511
```

You can find the full diff in the `full_diffs_list` array. If you index this array it will give you every guess for a given subkey (for example `full_diffs_list[5]` is the 5th subkey guess outputs).

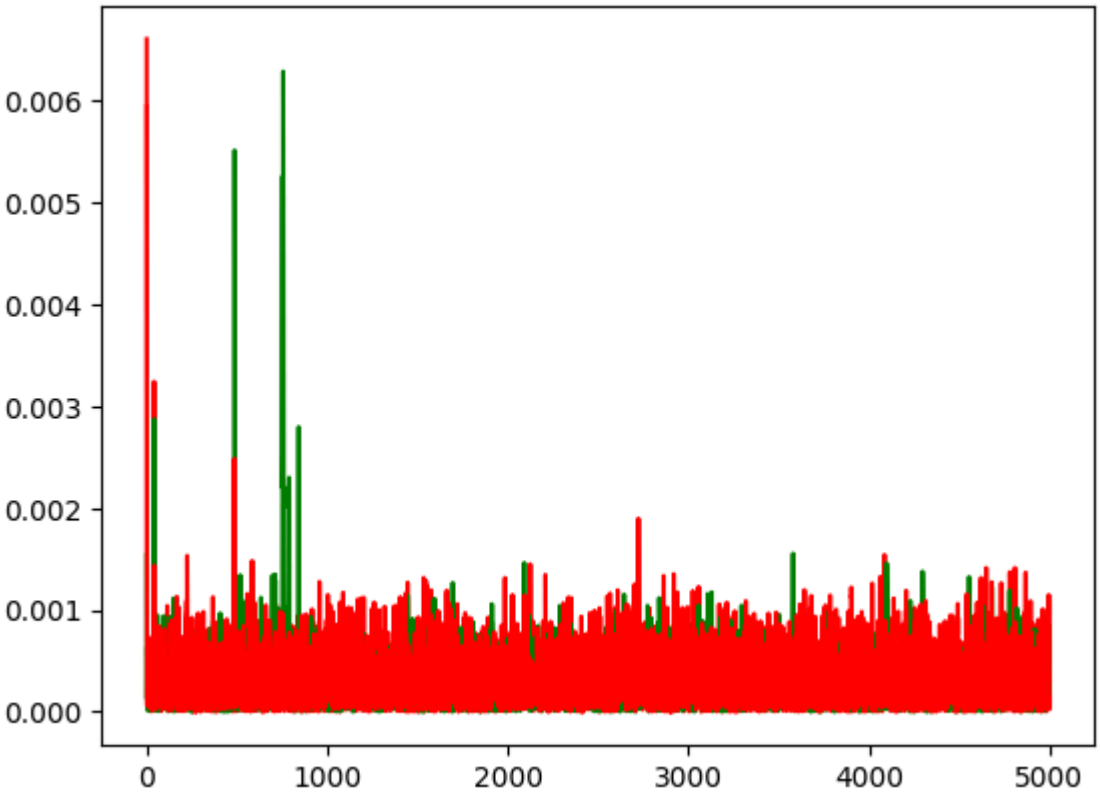
Using `full_diffs_list[N]` to get your selected subkey, plot the correct key by plotting `full_diffs_list[N][0xCORRECT]` in green as the *last* (so it appears on top). Plot a few other highly ranked guesses before that. In my example, this would look like:

```
%matplotlib notebook
import matplotlib.pylab as plt
```

```
    plt.plot(full_diffs_list[5][0xC5], 'r')
    plt.plot(full_diffs_list[5][0xCB], 'r')
    plt.plot(full_diffs_list[5][0xAE], 'g')
```

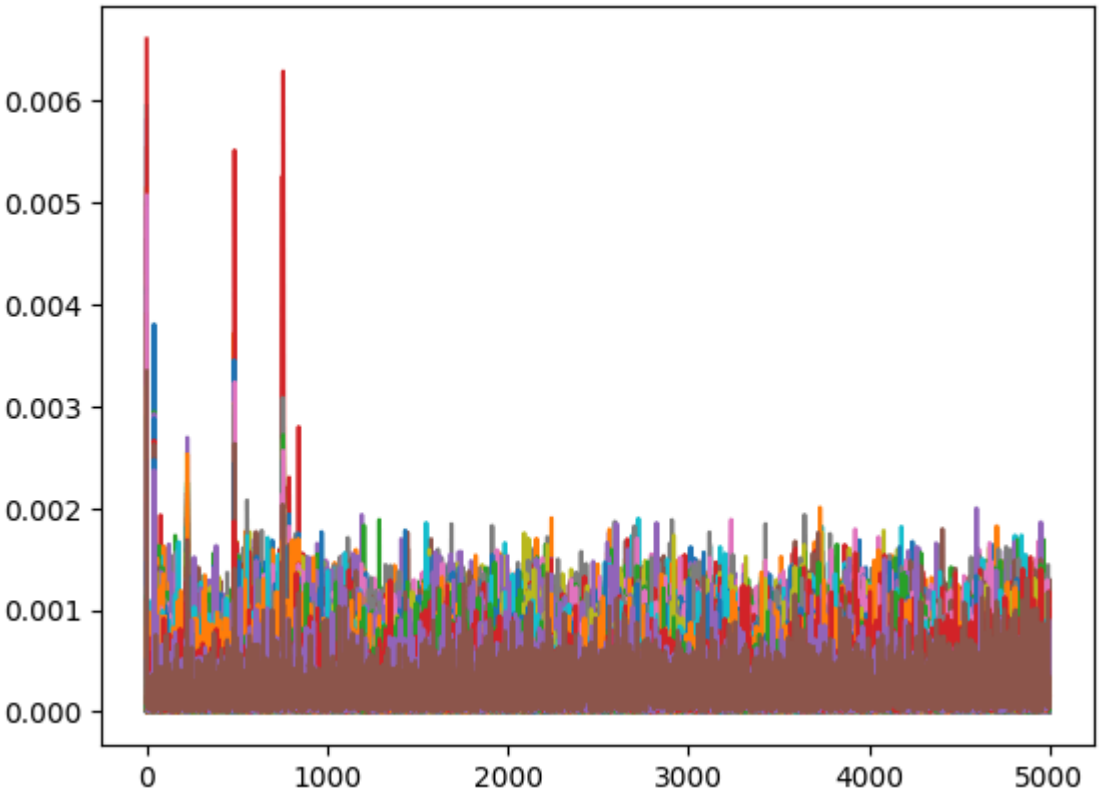
In [ ]:
```
# cw.plot(full_diffs_list[0][0x67]) * cw.plot(full_diffs_list[0][0x2B]) *
plt.plot(full_diffs_list[0][0x67], 'r')
plt.plot(full_diffs_list[0][0x2B], 'g')
plt.plot(full_diffs_list[0][0x09], 'r')
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x71e520c78160>]
```



Zoom in on the window, and you should notice there is a location where the correct peak is *higher* than the incorrect peaks. If you want to plot all the traces (this will get slow!) for a given trace, we can do so as the following:

In [ ]:
```
plt.figure()
subkey = 0
for guess in range(0, 256):
    plt.plot(full_diffs_list[subkey][guess])
plt.show()
```



Depending on your hardware, the previous may show a single nice large spike, or multiple large spikes. If we have the ghost peak problem you've probably got multiple spikes. The incorrect peaks may trail behind the correct locations -- we can first plot the correct locations by looking at the known key. The following will do that:

In [ ]:
```
fig = cw.plot()
for subkey in range(0, 16):
    fig *= cw.plot(full_diffs_list[subkey][known_key[subkey]])
fig
```

Out[ ]:

Windowing Peaks

The final trick here - see if there is some way to "window" the data that could be useful. For example, looking at the peaks you might notice that the correct peaks are always coming at 60 cycle offsets, with the first peak around sample 1100 (these will be different for your hardware).

So we could modify the loop to only look at differences after this point:

```
    for guess in range(0, 256):
        full_diff_trace = calculate_diffs(guess, subkey,
    bitnum)
        full_diff_trace = full_diff_trace[(1010 + subkey*60):]
        max_diffs[guess] = np.max(full_diff_trace)
        full_diffs[guess] = full_diff_trace
```

Copy the full DPA attack here - and try it out! See if you can get the correct key to come out for every byte.

In [ ]:
```
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,

#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_gues

    #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]))

    print("\n")
```

```
/tmp/ipykernel_6444/1681731736.py:13: TqdmDeprecationWarning: Please use `
tqdm.notebook.trange` instead of `tqdm.tnrange`
  for subkey in tnrange(0, 16, desc="Attacking Subkey"):
Attacking Subkey:   0%|          | 0/16 [00:00<?, ?it/s]
```

```
Subkey  0 - most likely 67 (actual 2B)
 Top 5 guesses:
   67 - Diff = 0.006610
   2B - Diff = 0.006290
   09 - Diff = 0.005949
   06 - Diff = 0.005807
   4D - Diff = 0.005537


Subkey  1 - most likely 9B (actual 7E)
 Top 5 guesses:
   9B - Diff = 0.006060
   AB - Diff = 0.005541
   96 - Diff = 0.005286
   3C - Diff = 0.005222
   74 - Diff = 0.005081


Subkey  2 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.006528
   5C - Diff = 0.005530
   B8 - Diff = 0.005272
   D3 - Diff = 0.005191
   C0 - Diff = 0.005077


Subkey  3 - most likely 52 (actual 16)
 Top 5 guesses:
   52 - Diff = 0.007875
   0B - Diff = 0.007829
   16 - Diff = 0.006548
   D1 - Diff = 0.006106
   A4 - Diff = 0.006043


Subkey  4 - most likely 49 (actual 28)
 Top 5 guesses:
   49 - Diff = 0.007173
   28 - Diff = 0.006761
   6C - Diff = 0.005939
   61 - Diff = 0.005384
   ED - Diff = 0.005312


Subkey  5 - most likely 73 (actual AE)
 Top 5 guesses:
   73 - Diff = 0.007094
   AE - Diff = 0.006200
   A5 - Diff = 0.006185
   BA - Diff = 0.006017
   1F - Diff = 0.005923


Subkey  6 - most likely D2 (actual D2)
 Top 5 guesses:
   D2 - Diff = 0.006811
   93 - Diff = 0.006074
   B6 - Diff = 0.005806
   12 - Diff = 0.005723
```

```
   70 - Diff = 0.005685


Subkey  7 - most likely 30 (actual A6)
 Top 5 guesses:
   30 - Diff = 0.006136
   14 - Diff = 0.005966
   90 - Diff = 0.005950
   77 - Diff = 0.005676
   7D - Diff = 0.005672


Subkey  8 - most likely AB (actual AB)
 Top 5 guesses:
   AB - Diff = 0.005828
   77 - Diff = 0.005724
   F0 - Diff = 0.005594
   67 - Diff = 0.005360
   B4 - Diff = 0.005099


Subkey  9 - most likely F7 (actual F7)
 Top 5 guesses:
   F7 - Diff = 0.005889
   51 - Diff = 0.005173
   C4 - Diff = 0.005091
   0D - Diff = 0.004821
   41 - Diff = 0.004818


Subkey 10 - most likely 16 (actual 15)
 Top 5 guesses:
   16 - Diff = 0.007881
   7B - Diff = 0.007756
   45 - Diff = 0.007000
   F0 - Diff = 0.006323
   15 - Diff = 0.006109


Subkey 11 - most likely F8 (actual 88)
 Top 5 guesses:
   F8 - Diff = 0.007431
   88 - Diff = 0.006527
   AB - Diff = 0.006402
   8A - Diff = 0.006398
   A7 - Diff = 0.005581


Subkey 12 - most likely BC (actual 09)
 Top 5 guesses:
   BC - Diff = 0.006620
   13 - Diff = 0.005459
   09 - Diff = 0.005410
   35 - Diff = 0.005366
   41 - Diff = 0.005346


Subkey 13 - most likely D3 (actual CF)
 Top 5 guesses:
   D3 - Diff = 0.006179
```

```
  CB - Diff = 0.005933
  C4 - Diff = 0.005719
  CF - Diff = 0.005707
  E3 - Diff = 0.005242


Subkey 14 - most likely 29 (actual 4F)
 Top 5 guesses:
  29 - Diff = 0.006403
  4F - Diff = 0.006189
  11 - Diff = 0.006081
  7C - Diff = 0.005913
  6C - Diff = 0.005697


Subkey 15 - most likely 96 (actual 3C)
 Top 5 guesses:
  96 - Diff = 0.006832
  A2 - Diff = 0.006397
  8A - Diff = 0.006342
  FB - Diff = 0.006278
  1A - Diff = 0.006218
```

Attacking Other Bits

So far we only looked at bit 0 $-$ but there are more bits involved here! You can first just try another bit that might be present, maybe they simply work better?

But you can also combine multiple bits by creating a most likely solution that applies across *all* bits.

The first one is easy to try out, as we defined the bit to attack in the previous script

The second option is a little more advanced. You can give it a try $-$ but in practice, if you are trying to combine multiple bits, a more effective method called the CPA attack will be used.

Conclusions & Next Steps

You've now seen how a DPA attack be be performed using a basic Python script. We'll experience much more effective attacks once we look at the CPA attack.

If you want to perform these attacks in practice, the Python code here isn't the most efficient! We'll look at faster options in later courses.

Tutorials derived from our open-source work must be released under the associated open-source license, and notice of the source must be *clearly displayed*. Only original copyright holders may license or authorize other distribution - while NewAE Technology Inc. holds the copyright for many tutorials, the github repository includes community contributions which we cannot license under special terms and **must** be maintained as an open-source release. Please contact us for special permissions (where possible).