Lab 3_3 - DPA on Firmware Implementation of AES (MAIN)

April 17, 2025

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

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.

1.1 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).

1.2 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:

[21]:
```
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]

def get_bit(data, bit):
    if data & (1<<bit):
        return 1
    else:
        return 0
```

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

[14]:
```
#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!
```

1.3 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
```

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.

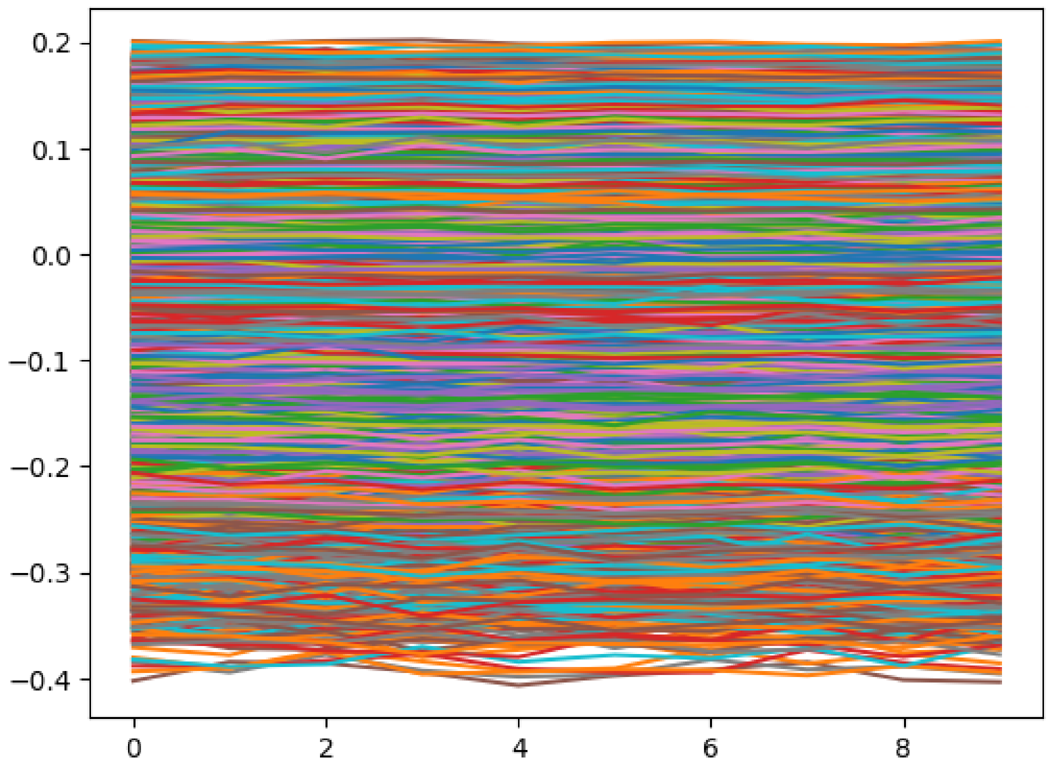
Be sure you get the `" OK to continue!"` print once you run the next cell, otherwise things will fail later on!

[15]:
```
from cwtraces import sca101_lab_data
import chipwhisperer as cw

data = sca101_lab_data["lab3_3"]()
trace_array =  data["trace_array"]
textin_array = data["textin_array"]

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:

[16]:
```
import matplotlib.pyplot as plt

plt.plot(trace_array[0:10])
plt.show()
```



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

[17]:
```
print(textin_array[0])
print(textin_array[1])
print(len(textin_array))
```

```
[114  66  40 199  69 192 177  78  97  85 186  45  62  36 126   6]
[160 118  77  45 232 113  55 110 207 203  94 123 132 217 169 251]
2500
```

1.4 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:

[18]:
```
import numpy as np

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`.

[35]:
```python
guess_byte = 0
key_values = [i for i in range(0,256)]
traces_index = [j for j in range(0,numtraces)]
mean_diffs = [0]*256

for guess_key_value in key_values :
    one_list = []
    zero_list = []

    for trace_index in traces_index :
        input_byte = textin_array[trace_index][guess_byte]

        hypothetical_leakage = aes_internal(input_byte, guess_key_value)

        if hypothetical_leakage & 0x01 :
            one_list.append(trace_array[trace_index])
        else :
            zero_list.append(trace_array[trace_index])
    one_avg = np.mean(one_list, axis=0)
    zero_avg = np.mean(zero_list, axis=0)

    max_diff_value = abs(one_avg - zero_avg)
    mean_diffs.append(np.max(abs(one_avg - zero_avg)))

indexes = np.argsort(mean_diffs)[::-1][0:5]
for index in indexes :
    print("Guessing :", hex(key_values[index]), ' : ', mean_diffs[index])
```

```
---------------------------------------------------------------------------
```

```
IndexError                                Traceback (most recent call last)
Cell In[35], line 27
     25 indexes = np.argsort(mean_diffs)[::-1][0:5]
     26 for index in indexes :
---> 27     print("Guessing :", hex(key_values[index]), ' : ', mean_diffs[index])

IndexError: list index out of range
```

1.4.1 Hint 1: General Program Flow

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

[ ]:
```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
```

1.4.2 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.

[25]:
```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][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)
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][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)
mean_diffs_ff = np.max(abs(one_avg - zero_avg))

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

```
Max SAD for 0x2B: 0.002539628236491495
Max SAD for 0xFF: 0.0008722838474198441
```

1.5 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]`:

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

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.

1.6 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):

```
[38]: 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)
```

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:

[39]:
```
cw.plot(calculate_diffs(0x2B)) * cw.plot(calculate_diffs(0x2C)) * cw.
 ↪plot(calculate_diffs(0x2D))
```

[39]:
```
:Overlay
   .Curve.I   :Curve   [x]   (y)
   .Curve.II  :Curve   [x]   (y)
   .Curve.III :Curve   [x]   (y)
```

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

1.7 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!
```

[41]:
```
bytes_found = []

for guess_byte in range(16):
    mean_diffs = [0]*256
    key_values = [i for i in range(0,256)]
    traces_index = [j for j in range(0,numtraces)]

    for guess_key_value in key_values :
        one_list = []
        zero_list = []

        for trace_index in traces_index :
            input_byte = textin_array[trace_index][guess_byte]

            hypothetical_leakage = aes_internal(input_byte, guess_key_value)

            if hypothetical_leakage & 0x01 :
                one_list.append(trace_array[trace_index])
            else :
                zero_list.append(trace_array[trace_index])

        one_avg = np.mean(one_list, axis=0)
        zero_avg = np.mean(zero_list, axis=0)

        max_diff_value = abs(one_avg - zero_avg)
        mean_diffs.append(np.max(abs(one_avg - zero_avg)))

    indexes = np.argsort(mean_diffs)[::-1][0:5]

    # keep best guess
    bytes_found.append(indexes[0])

    print(hex(indexes[0]), ":", mean_diffs[indexes[0]])
```

```
0x12b : 0.002539628236491495
0x17e : 0.002434996796679967
0x115 : 0.00238105965719366
0x116 : 0.0026789590062572914
0x128 : 0.0021884711861236486
0x1ae : 0.0023793848040562747
0x1d2 : 0.002673822985760881
0x1a6 : 0.002340922005064078
0x1ab : 0.0023747873294555667
0x1f7 : 0.002366572945686485
0x115 : 0.0022186389310675098
0x188 : 0.002506421589613242
0x109 : 0.0021184650706820435
0x1cf : 0.0019617221625988918
0x14f : 0.002342071495008751
0x13c : 0.002632352336828627
```

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).

1.8 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):

[43]:
```
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, 0x15,␣
 ↪0x88, 0x09, 0xcf, 0x4f, 0x3c]

#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_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]))

    print("\n")
```

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

Subkey  0 - most likely 2B (actual 2B)
 Top 5 guesses:
   2B - Diff = 0.002540
   69 - Diff = 0.001297
   40 - Diff = 0.001297
   75 - Diff = 0.001285
   97 - Diff = 0.001263


Subkey  1 - most likely 7E (actual 7E)
 Top 5 guesses:
   7E - Diff = 0.002435
   15 - Diff = 0.001535
   CF - Diff = 0.001460
   8B - Diff = 0.001438
   3C - Diff = 0.001420


Subkey  2 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.002381
   8E - Diff = 0.001424
   98 - Diff = 0.001399
   CA - Diff = 0.001386
   F5 - Diff = 0.001351


Subkey  3 - most likely 16 (actual 16)
 Top 5 guesses:
   16 - Diff = 0.002679
   A9 - Diff = 0.001557
   AB - Diff = 0.001543
   DC - Diff = 0.001415
   C0 - Diff = 0.001375
```

```
Subkey  4 - most likely 28 (actual 28)
 Top 5 guesses:
   28 - Diff = 0.002188
   CB - Diff = 0.001552
   43 - Diff = 0.001452
   95 - Diff = 0.001266
   9C - Diff = 0.001263


Subkey  5 - most likely AE (actual AE)
 Top 5 guesses:
   AE - Diff = 0.002379
   13 - Diff = 0.001710
   7A - Diff = 0.001418
   37 - Diff = 0.001339
   B7 - Diff = 0.001290


Subkey  6 - most likely D2 (actual D2)
 Top 5 guesses:
   D2 - Diff = 0.002674
   6F - Diff = 0.001594
   4B - Diff = 0.001344
   E1 - Diff = 0.001339
   B9 - Diff = 0.001334


Subkey  7 - most likely A6 (actual A6)
 Top 5 guesses:
   A6 - Diff = 0.002341
   73 - Diff = 0.001466
   CD - Diff = 0.001361
   22 - Diff = 0.001310
   B9 - Diff = 0.001289


Subkey  8 - most likely AB (actual AB)
 Top 5 guesses:
   AB - Diff = 0.002375
   C0 - Diff = 0.001509
   E9 - Diff = 0.001367
   46 - Diff = 0.001359
   B6 - Diff = 0.001354


Subkey  9 - most likely F7 (actual F7)
 Top 5 guesses:
   F7 - Diff = 0.002367
   09 - Diff = 0.001455
   B5 - Diff = 0.001435
   07 - Diff = 0.001421
   C9 - Diff = 0.001334


Subkey 10 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.002219
   8C - Diff = 0.001350
   89 - Diff = 0.001326
   43 - Diff = 0.001324
   D3 - Diff = 0.001296


Subkey 11 - most likely 88 (actual 88)
 Top 5 guesses:
   88 - Diff = 0.002506
   11 - Diff = 0.001517
   6A - Diff = 0.001351
   0C - Diff = 0.001299
   96 - Diff = 0.001289


Subkey 12 - most likely 09 (actual 09)
 Top 5 guesses:
   09 - Diff = 0.002118
   9D - Diff = 0.001399
   FF - Diff = 0.001370
   1D - Diff = 0.001290
   6C - Diff = 0.001258


Subkey 13 - most likely CF (actual CF)
 Top 5 guesses:
   CF - Diff = 0.001962
   74 - Diff = 0.001449
   23 - Diff = 0.001404
   FD - Diff = 0.001397
   1F - Diff = 0.001393


Subkey 14 - most likely 4F (actual 4F)
 Top 5 guesses:
   4F - Diff = 0.002342
   D6 - Diff = 0.001710
   24 - Diff = 0.001607
   0C - Diff = 0.001373
   0D - Diff = 0.001366


Subkey 15 - most likely 3C (actual 3C)
 Top 5 guesses:
   3C - Diff = 0.002632
   81 - Diff = 0.001800
   57 - Diff = 0.001423
   7E - Diff = 0.001417
   15 - Diff = 0.001288
```

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).

1.8.1 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
```

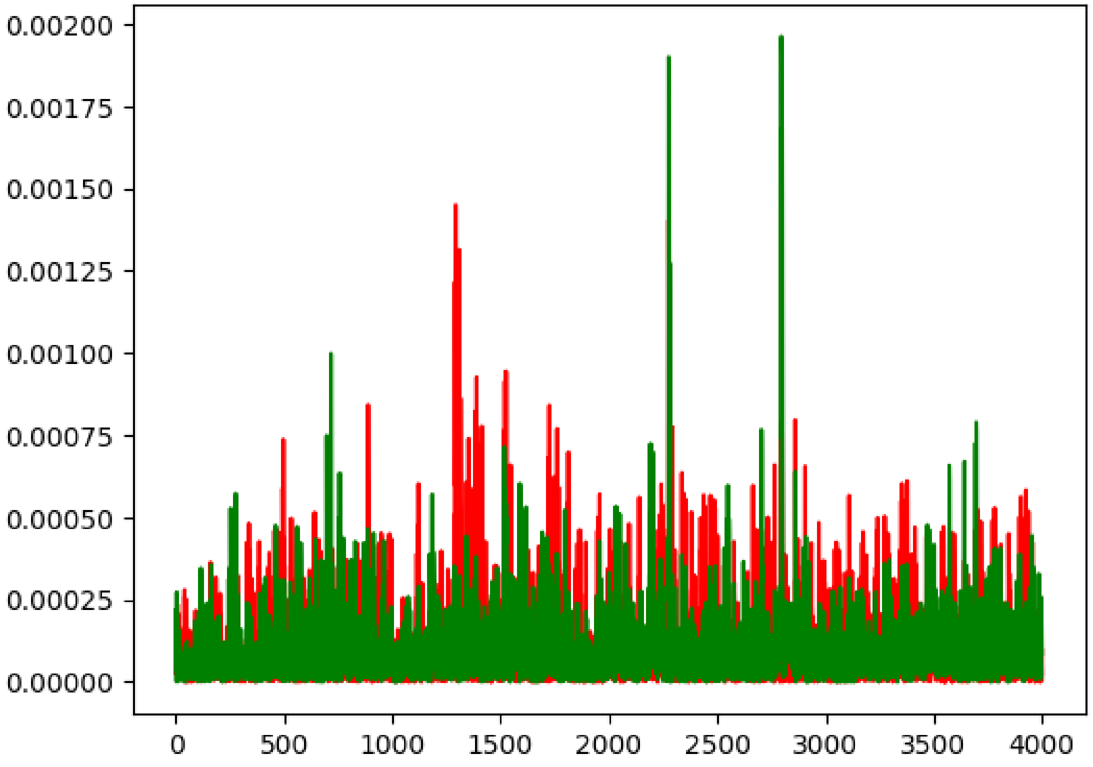
You can find the full diff in the `full_diffs_list` array. If you index this array it will give you every guess for a given subkey (for example `full_diffs_list[5]` is the 5th subkey guess outputs).

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')
```

[44]:
```
plt.plot(full_diffs_list[13][0x74], 'r')
plt.plot(full_diffs_list[13][0x23], 'r')
plt.plot(full_diffs_list[13][0xCF], 'g')
```

```
[44]: [<matplotlib.lines.Line2D at 0x12fb7b0d0>]
```



```
[ ]: cw.plot(full_diffs_list[5][0xC5]) * cw.plot(full_diffs_list[5][0xCB]) * cw.
     ↪plot(full_diffs_list[5][0xAE])
```

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:

```
[45]: fig = cw.plot()
      subkey = 13
      for guess in range(0, 256):
          fig *= cw.plot(full_diffs_list[subkey][guess])
      fig
```

```
[45]: :Overlay
         .Curve.I          :Curve   [x]   (y)
         .Curve.II         :Curve   [x]   (y)
         .Curve.III        :Curve   [x]   (y)
         .Curve.IV         :Curve   [x]   (y)
         .Curve.V          :Curve   [x]   (y)
         .Curve.VI         :Curve   [x]   (y)
```

```
.Curve.VII       :Curve   [x]   (y)
.Curve.VIII      :Curve   [x]   (y)
.Curve.IX        :Curve   [x]   (y)
.Curve.X         :Curve   [x]   (y)
.Curve.XI        :Curve   [x]   (y)
.Curve.XII       :Curve   [x]   (y)
.Curve.XIII      :Curve   [x]   (y)
.Curve.XIV       :Curve   [x]   (y)
.Curve.XV        :Curve   [x]   (y)
.Curve.XVI       :Curve   [x]   (y)
.Curve.XVII      :Curve   [x]   (y)
.Curve.XVIII     :Curve   [x]   (y)
.Curve.XIX       :Curve   [x]   (y)
.Curve.XX        :Curve   [x]   (y)
.Curve.XXI       :Curve   [x]   (y)
.Curve.XXII      :Curve   [x]   (y)
.Curve.XXIII     :Curve   [x]   (y)
.Curve.XXIV      :Curve   [x]   (y)
.Curve.XXV       :Curve   [x]   (y)
.Curve.XXVI      :Curve   [x]   (y)
.Curve.XXVII     :Curve   [x]   (y)
.Curve.XXVIII    :Curve   [x]   (y)
.Curve.XXIX      :Curve   [x]   (y)
.Curve.XXX       :Curve   [x]   (y)
.Curve.XXXI      :Curve   [x]   (y)
.Curve.XXXII     :Curve   [x]   (y)
.Curve.XXXIII    :Curve   [x]   (y)
.Curve.XXXIV     :Curve   [x]   (y)
.Curve.XXXV      :Curve   [x]   (y)
.Curve.XXXVI     :Curve   [x]   (y)
.Curve.XXXVII    :Curve   [x]   (y)
.Curve.XXXVIII   :Curve   [x]   (y)
.Curve.XXXIX     :Curve   [x]   (y)
.Curve.XL        :Curve   [x]   (y)
.Curve.XLI       :Curve   [x]   (y)
.Curve.XLII      :Curve   [x]   (y)
.Curve.XLIII     :Curve   [x]   (y)
.Curve.XLIV      :Curve   [x]   (y)
.Curve.XLV       :Curve   [x]   (y)
.Curve.XLVI      :Curve   [x]   (y)
.Curve.XLVII     :Curve   [x]   (y)
.Curve.XLVIII    :Curve   [x]   (y)
.Curve.XLIX      :Curve   [x]   (y)
.Curve.L         :Curve   [x]   (y)
.Curve.LI        :Curve   [x]   (y)
.Curve.LII       :Curve   [x]   (y)
.Curve.LIII      :Curve   [x]   (y)
```

```
.Curve.LIV       :Curve   [x]   (y)
.Curve.LV        :Curve   [x]   (y)
.Curve.LVI       :Curve   [x]   (y)
.Curve.LVII      :Curve   [x]   (y)
.Curve.LVIII     :Curve   [x]   (y)
.Curve.LIX       :Curve   [x]   (y)
.Curve.LX        :Curve   [x]   (y)
.Curve.LXI       :Curve   [x]   (y)
.Curve.LXII      :Curve   [x]   (y)
.Curve.LXIII     :Curve   [x]   (y)
.Curve.LXIV      :Curve   [x]   (y)
.Curve.LXV       :Curve   [x]   (y)
.Curve.LXVI      :Curve   [x]   (y)
.Curve.LXVII     :Curve   [x]   (y)
.Curve.LXVIII    :Curve   [x]   (y)
.Curve.LXIX      :Curve   [x]   (y)
.Curve.LXX       :Curve   [x]   (y)
.Curve.LXXI      :Curve   [x]   (y)
.Curve.LXXII     :Curve   [x]   (y)
.Curve.LXXIII    :Curve   [x]   (y)
.Curve.LXXIV     :Curve   [x]   (y)
.Curve.LXXV      :Curve   [x]   (y)
.Curve.LXXVI     :Curve   [x]   (y)
.Curve.LXXVII    :Curve   [x]   (y)
.Curve.LXXVIII   :Curve   [x]   (y)
.Curve.LXXIX     :Curve   [x]   (y)
.Curve.LXXX      :Curve   [x]   (y)
.Curve.LXXXI     :Curve   [x]   (y)
.Curve.LXXXII    :Curve   [x]   (y)
.Curve.LXXXIII   :Curve   [x]   (y)
.Curve.LXXXIV    :Curve   [x]   (y)
.Curve.LXXXV     :Curve   [x]   (y)
.Curve.LXXXVI    :Curve   [x]   (y)
.Curve.LXXXVII   :Curve   [x]   (y)
.Curve.LXXXVIII  :Curve   [x]   (y)
.Curve.LXXXIX    :Curve   [x]   (y)
.Curve.XC        :Curve   [x]   (y)
.Curve.XCI       :Curve   [x]   (y)
.Curve.XCII      :Curve   [x]   (y)
.Curve.XCIII     :Curve   [x]   (y)
.Curve.XCIV      :Curve   [x]   (y)
.Curve.XCV       :Curve   [x]   (y)
.Curve.XCVI      :Curve   [x]   (y)
.Curve.XCVII     :Curve   [x]   (y)
.Curve.XCVIII    :Curve   [x]   (y)
.Curve.XCIX      :Curve   [x]   (y)
.Curve.C         :Curve   [x]   (y)
```

```
.Curve.CI         :Curve   [x]   (y)
.Curve.CII        :Curve   [x]   (y)
.Curve.CIII       :Curve   [x]   (y)
.Curve.CIV        :Curve   [x]   (y)
.Curve.CV         :Curve   [x]   (y)
.Curve.CVI        :Curve   [x]   (y)
.Curve.CVII       :Curve   [x]   (y)
.Curve.CVIII      :Curve   [x]   (y)
.Curve.CIX        :Curve   [x]   (y)
.Curve.CX         :Curve   [x]   (y)
.Curve.CXI        :Curve   [x]   (y)
.Curve.CXII       :Curve   [x]   (y)
.Curve.CXIII      :Curve   [x]   (y)
.Curve.CXIV       :Curve   [x]   (y)
.Curve.CXV        :Curve   [x]   (y)
.Curve.CXVI       :Curve   [x]   (y)
.Curve.CXVII      :Curve   [x]   (y)
.Curve.CXVIII     :Curve   [x]   (y)
.Curve.CXIX       :Curve   [x]   (y)
.Curve.CXX        :Curve   [x]   (y)
.Curve.CXXI       :Curve   [x]   (y)
.Curve.CXXII      :Curve   [x]   (y)
.Curve.CXXIII     :Curve   [x]   (y)
.Curve.CXXIV      :Curve   [x]   (y)
.Curve.CXXV       :Curve   [x]   (y)
.Curve.CXXVI      :Curve   [x]   (y)
.Curve.CXXVII     :Curve   [x]   (y)
.Curve.CXXVIII    :Curve   [x]   (y)
.Curve.CXXIX      :Curve   [x]   (y)
.Curve.CXXX       :Curve   [x]   (y)
.Curve.CXXXI      :Curve   [x]   (y)
.Curve.CXXXII     :Curve   [x]   (y)
.Curve.CXXXIII    :Curve   [x]   (y)
.Curve.CXXXIV     :Curve   [x]   (y)
.Curve.CXXXV      :Curve   [x]   (y)
.Curve.CXXXVI     :Curve   [x]   (y)
.Curve.CXXXVII    :Curve   [x]   (y)
.Curve.CXXXVIII   :Curve   [x]   (y)
.Curve.CXXXIX     :Curve   [x]   (y)
.Curve.CXL        :Curve   [x]   (y)
.Curve.CXLI       :Curve   [x]   (y)
.Curve.CXLII      :Curve   [x]   (y)
.Curve.CXLIII     :Curve   [x]   (y)
.Curve.CXLIV      :Curve   [x]   (y)
.Curve.CXLV       :Curve   [x]   (y)
.Curve.CXLVI      :Curve   [x]   (y)
.Curve.CXLVII     :Curve   [x]   (y)
```

```
.Curve.CXLVIII   :Curve   [x]   (y)
.Curve.CXLIX     :Curve   [x]   (y)
.Curve.CL        :Curve   [x]   (y)
.Curve.CLI       :Curve   [x]   (y)
.Curve.CLII      :Curve   [x]   (y)
.Curve.CLIII     :Curve   [x]   (y)
.Curve.CLIV      :Curve   [x]   (y)
.Curve.CLV       :Curve   [x]   (y)
.Curve.CLVI      :Curve   [x]   (y)
.Curve.CLVII     :Curve   [x]   (y)
.Curve.CLVIII    :Curve   [x]   (y)
.Curve.CLIX      :Curve   [x]   (y)
.Curve.CLX       :Curve   [x]   (y)
.Curve.CLXI      :Curve   [x]   (y)
.Curve.CLXII     :Curve   [x]   (y)
.Curve.CLXIII    :Curve   [x]   (y)
.Curve.CLXIV     :Curve   [x]   (y)
.Curve.CLXV      :Curve   [x]   (y)
.Curve.CLXVI     :Curve   [x]   (y)
.Curve.CLXVII    :Curve   [x]   (y)
.Curve.CLXVIII   :Curve   [x]   (y)
.Curve.CLXIX     :Curve   [x]   (y)
.Curve.CLXX      :Curve   [x]   (y)
.Curve.CLXXI     :Curve   [x]   (y)
.Curve.CLXXII    :Curve   [x]   (y)
.Curve.CLXXIII   :Curve   [x]   (y)
.Curve.CLXXIV    :Curve   [x]   (y)
.Curve.CLXXV     :Curve   [x]   (y)
.Curve.CLXXVI    :Curve   [x]   (y)
.Curve.CLXXVII   :Curve   [x]   (y)
.Curve.CLXXVIII  :Curve   [x]   (y)
.Curve.CLXXIX    :Curve   [x]   (y)
.Curve.CLXXX     :Curve   [x]   (y)
.Curve.CLXXXI    :Curve   [x]   (y)
.Curve.CLXXXII   :Curve   [x]   (y)
.Curve.CLXXXIII  :Curve   [x]   (y)
.Curve.CLXXXIV   :Curve   [x]   (y)
.Curve.CLXXXV    :Curve   [x]   (y)
.Curve.CLXXXVI   :Curve   [x]   (y)
.Curve.CLXXXVII  :Curve   [x]   (y)
.Curve.CLXXXVIII :Curve   [x]   (y)
.Curve.CLXXXIX   :Curve   [x]   (y)
.Curve.CXC       :Curve   [x]   (y)
.Curve.CXCI      :Curve   [x]   (y)
.Curve.CXCII     :Curve   [x]   (y)
.Curve.CXCIII    :Curve   [x]   (y)
.Curve.CXCIV     :Curve   [x]   (y)
```

```
.Curve.CXCV      :Curve   [x]   (y)
.Curve.CXCVI     :Curve   [x]   (y)
.Curve.CXCVII    :Curve   [x]   (y)
.Curve.CXCVIII   :Curve   [x]   (y)
.Curve.CXCIX     :Curve   [x]   (y)
.Curve.CC        :Curve   [x]   (y)
.Curve.CCI       :Curve   [x]   (y)
.Curve.CCII      :Curve   [x]   (y)
.Curve.CCIII     :Curve   [x]   (y)
.Curve.CCIV      :Curve   [x]   (y)
.Curve.CCV       :Curve   [x]   (y)
.Curve.CCVI      :Curve   [x]   (y)
.Curve.CCVII     :Curve   [x]   (y)
.Curve.CCVIII    :Curve   [x]   (y)
.Curve.CCIX      :Curve   [x]   (y)
.Curve.CCX       :Curve   [x]   (y)
.Curve.CCXI      :Curve   [x]   (y)
.Curve.CCXII     :Curve   [x]   (y)
.Curve.CCXIII    :Curve   [x]   (y)
.Curve.CCXIV     :Curve   [x]   (y)
.Curve.CCXV      :Curve   [x]   (y)
.Curve.CCXVI     :Curve   [x]   (y)
.Curve.CCXVII    :Curve   [x]   (y)
.Curve.CCXVIII   :Curve   [x]   (y)
.Curve.CCXIX     :Curve   [x]   (y)
.Curve.CCXX      :Curve   [x]   (y)
.Curve.CCXXI     :Curve   [x]   (y)
.Curve.CCXXII    :Curve   [x]   (y)
.Curve.CCXXIII   :Curve   [x]   (y)
.Curve.CCXXIV    :Curve   [x]   (y)
.Curve.CCXXV     :Curve   [x]   (y)
.Curve.CCXXVI    :Curve   [x]   (y)
.Curve.CCXXVII   :Curve   [x]   (y)
.Curve.CCXXVIII  :Curve   [x]   (y)
.Curve.CCXXIX    :Curve   [x]   (y)
.Curve.CCXXX     :Curve   [x]   (y)
.Curve.CCXXXI    :Curve   [x]   (y)
.Curve.CCXXXII   :Curve   [x]   (y)
.Curve.CCXXXIII  :Curve   [x]   (y)
.Curve.CCXXXIV   :Curve   [x]   (y)
.Curve.CCXXXV    :Curve   [x]   (y)
.Curve.CCXXXVI   :Curve   [x]   (y)
.Curve.CCXXXVII  :Curve   [x]   (y)
.Curve.CCXXXVIII :Curve   [x]   (y)
.Curve.CCXXXIX   :Curve   [x]   (y)
.Curve.CCXL      :Curve   [x]   (y)
.Curve.CCXLI     :Curve   [x]   (y)
```

```
    .Curve.CCXLII    :Curve   [x]   (y)
    .Curve.CCXLIII   :Curve   [x]   (y)
    .Curve.CCXLIV    :Curve   [x]   (y)
    .Curve.CCXLV     :Curve   [x]   (y)
    .Curve.CCXLVI    :Curve   [x]   (y)
    .Curve.CCXLVII   :Curve   [x]   (y)
    .Curve.CCXLVIII  :Curve   [x]   (y)
    .Curve.CCXLIX    :Curve   [x]   (y)
    .Curve.CCL       :Curve   [x]   (y)
    .Curve.CCLI      :Curve   [x]   (y)
    .Curve.CCLII     :Curve   [x]   (y)
    .Curve.CCLIII    :Curve   [x]   (y)
    .Curve.CCLIV     :Curve   [x]   (y)
    .Curve.CCLV      :Curve   [x]   (y)
    .Curve.CCLVI     :Curve   [x]   (y)
    .Curve.CCLVII    :Curve   [x]   (y)
```

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:

[46]:
```python
fig = cw.plot()
for subkey in range(0, 16):
    fig *= cw.plot(full_diffs_list[subkey][known_key[subkey]])
fig
```

[46]:
```
:Overlay
    .Curve.I     :Curve   [x]   (y)
    .Curve.II    :Curve   [x]   (y)
    .Curve.III   :Curve   [x]   (y)
    .Curve.IV    :Curve   [x]   (y)
    .Curve.V     :Curve   [x]   (y)
    .Curve.VI    :Curve   [x]   (y)
    .Curve.VII   :Curve   [x]   (y)
    .Curve.VIII  :Curve   [x]   (y)
    .Curve.IX    :Curve   [x]   (y)
    .Curve.X     :Curve   [x]   (y)
    .Curve.XI    :Curve   [x]   (y)
    .Curve.XII   :Curve   [x]   (y)
    .Curve.XIII  :Curve   [x]   (y)
    .Curve.XIV   :Curve   [x]   (y)
    .Curve.XV    :Curve   [x]   (y)
    .Curve.XVI   :Curve   [x]   (y)
    .Curve.XVII  :Curve   [x]   (y)
```

1.8.2 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.

[47]:
```
#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)
        full_diff_trace = full_diff_trace[(1010 + subkey*60):]
        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]))

    print("\n")
```

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

Subkey  0 - most likely 2B (actual 2B)
 Top 5 guesses:
   2B - Diff = 0.002540
   69 - Diff = 0.001297
   40 - Diff = 0.001297
   75 - Diff = 0.001285
   97 - Diff = 0.001263


Subkey  1 - most likely 7E (actual 7E)
 Top 5 guesses:
   7E - Diff = 0.002435
   15 - Diff = 0.001535
   CF - Diff = 0.001460
   8B - Diff = 0.001438
   3C - Diff = 0.001420


Subkey  2 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.002381
   8E - Diff = 0.001424
   98 - Diff = 0.001399
   CA - Diff = 0.001386
   F5 - Diff = 0.001351


Subkey  3 - most likely 16 (actual 16)
 Top 5 guesses:
   16 - Diff = 0.002679
   A9 - Diff = 0.001557
   AB - Diff = 0.001543
   DC - Diff = 0.001415
   C0 - Diff = 0.001375


Subkey  4 - most likely 28 (actual 28)
 Top 5 guesses:
   28 - Diff = 0.002188
   43 - Diff = 0.001452
   95 - Diff = 0.001266
   BD - Diff = 0.001247
   10 - Diff = 0.001242


Subkey  5 - most likely AE (actual AE)
 Top 5 guesses:
   AE - Diff = 0.002379
   13 - Diff = 0.001710
   7A - Diff = 0.001418
   37 - Diff = 0.001339
   B7 - Diff = 0.001290


Subkey  6 - most likely D2 (actual D2)
 Top 5 guesses:
   D2 - Diff = 0.002674
   6F - Diff = 0.001594
   4B - Diff = 0.001344
   E1 - Diff = 0.001339
   B9 - Diff = 0.001334


Subkey  7 - most likely A6 (actual A6)
 Top 5 guesses:
   A6 - Diff = 0.002341
   CD - Diff = 0.001361
   22 - Diff = 0.001310
   B9 - Diff = 0.001289
   2B - Diff = 0.001289


Subkey  8 - most likely AB (actual AB)
 Top 5 guesses:
   AB - Diff = 0.002375
   C0 - Diff = 0.001509
   E9 - Diff = 0.001367
   46 - Diff = 0.001359
   E1 - Diff = 0.001352


Subkey  9 - most likely F7 (actual F7)
 Top 5 guesses:
   F7 - Diff = 0.002367
   09 - Diff = 0.001455
   B5 - Diff = 0.001435
   C9 - Diff = 0.001334
   1B - Diff = 0.001299
```

```
Subkey 10 - most likely 15 (actual 15)
 Top 5 guesses:
   15 - Diff = 0.002219
   8C - Diff = 0.001350
   D3 - Diff = 0.001296
   F9 - Diff = 0.001284
   5F - Diff = 0.001264


Subkey 11 - most likely 88 (actual 88)
 Top 5 guesses:
   88 - Diff = 0.002506
   11 - Diff = 0.001517
   6A - Diff = 0.001351
   0C - Diff = 0.001299
   96 - Diff = 0.001289


Subkey 12 - most likely 09 (actual 09)
 Top 5 guesses:
   09 - Diff = 0.002118
   9D - Diff = 0.001399
   FF - Diff = 0.001370
   6C - Diff = 0.001258
   37 - Diff = 0.001241


Subkey 13 - most likely CF (actual CF)
 Top 5 guesses:
   CF - Diff = 0.001962
   23 - Diff = 0.001404
   FD - Diff = 0.001397
   1F - Diff = 0.001393
   72 - Diff = 0.001350


Subkey 14 - most likely 4F (actual 4F)
 Top 5 guesses:
   4F - Diff = 0.002342
   D6 - Diff = 0.001710
   24 - Diff = 0.001607
   0C - Diff = 0.001373
   0D - Diff = 0.001366


Subkey 15 - most likely 3C (actual 3C)
```

```
Top 5 guesses:
  3C - Diff = 0.002632
  81 - Diff = 0.001800
  57 - Diff = 0.001423
  7E - Diff = 0.001417
  15 - Diff = 0.001288
```

1.8.3 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.

1.9 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.

[ ]: