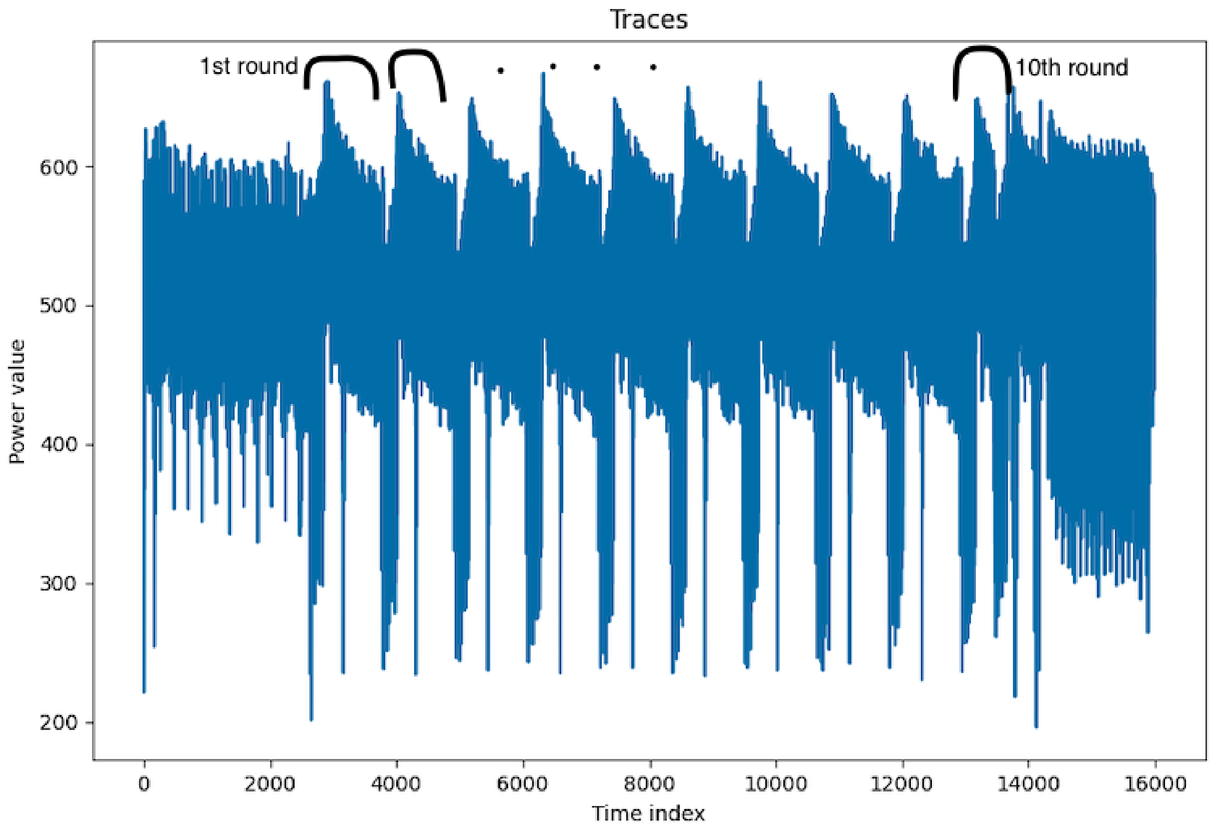
SCA_single_bit_DPA

Note: if you do it before the sbox you'd only be able to distinguish one key since sbox actually will use all 8bits of the key.

Single Bit DPA

Scenario:
We have encryption power consumption traces from AES operation applied to known plaintexts.
A trace looks like the following:



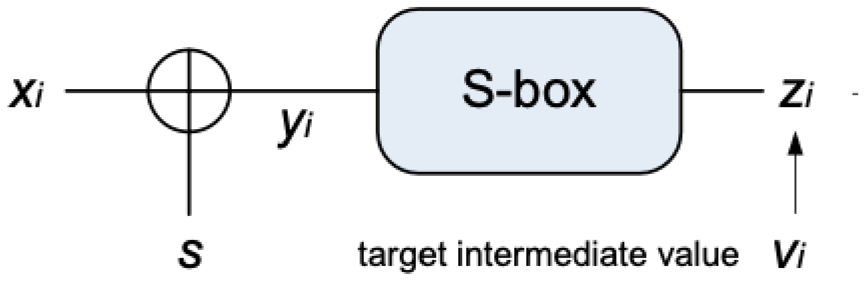
We can identify the 10 rounds of the AES, with the last round deviating from the other ones since it does not do the mix-columns operation.

We are going to attack the first S-box of AES. Our attack will use the known plaintexts and corresponding traces of these plaintext encryptions.

pre-processing

we remove the mean from each trace and take only the values that could correspond to the first s-box, which is between time samples 2600 to 3900.

S-box



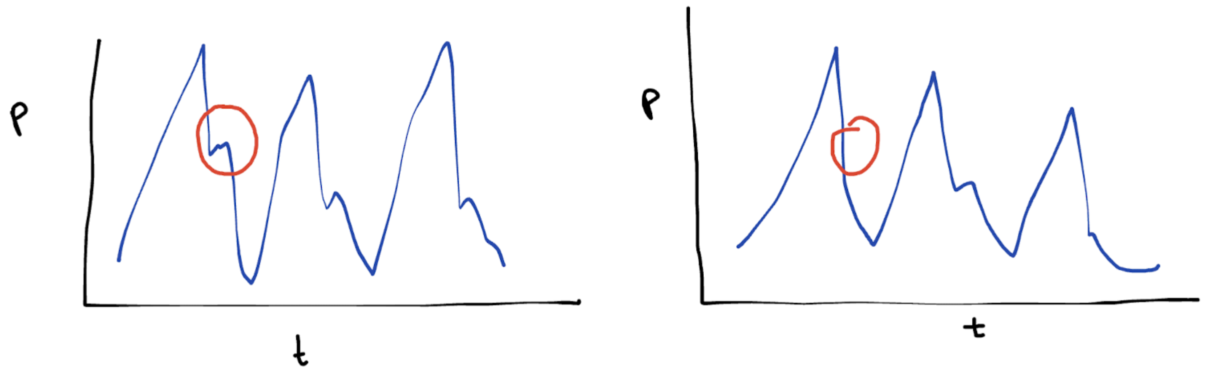
In this diagram x_i corresponds to the byte we are attacking, s is the key hypothesis (basically the key candidate) and z_i is the intermediate value we will calculate.

Note:

- A plaintext is 16 bytes
- A key is also 16 bytes
- So we would like to find each key byte, we will select each byte from the plaintext, as x_i and attack that with all possible keys (256 possibilities)

Then based on the last bit of z_i we are going to separate traces into 2 sets. This was not immediately clear for me so let's try to explain.

Imagine two traces:



On the left one let's say the intermediate value has its LSB set to one, and the right side the intermediate value has its LSB set to 0. We would expect to see a small difference in traces since we have one bit less switching on and off and of course other effects it entails as well so this is just imaginary at the moment (CMOS transistor consuming less power since one bit less switch is required to process the value). But the point is the two traces will have a (small) difference based on the LSB of the intermediate value being 1 or 0.

Okay, we can wrap our head around that there will be difference in these traces, how can this be used?

This can be used in the following way:

- We have traces and corresponding plaintexts that are being encrypted.
- We know the S-box input output (bijection)
- We can predict the intermediary value by simply:

$$v_i^* = sbox[x_i \; xor \; s]$$

- Then we can use this intermediary value, look at its LSB and put the corresponding trace in one of the sets we are building for the given key guess.
  Putting these into code:

```python
def dpa_byte(index, pts, traces):
    """
    index: index of the byte on which to perform the attack.
    pts: the plaintext of each encryption performed.
    traces: the power measurements performed for each encryption.
(n_traces, n_samples)
    return: an np.array with the key bytes, from highest probable to
less probable (256,)
    """

    # TODO: Implement the function
    mean_diffs  = []
    for key_guess in range(256):
        set0 = []
        set1 = []
        for i in range(traces.shape[0]):
            x_i = pts[i,:][index] # get the byte of the plaintext
            v_i_star = sbox[x_i ^ key_guess] # apply the prediction
            trace = traces[i,:] # (16000,)
            if v_i_star % 2 == 1: # If LSB = 1 => modulo 2 returns 1
                set1.append(trace)
            else:
                set0.append(trace)
        # What to do with these sets
```

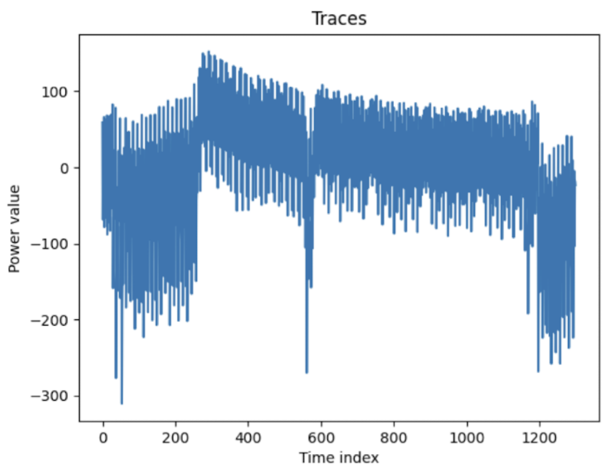
Okay, now we have these two sets. To repeat, for a given key guess s, we have used it to make a prediction of the intermediate value v_i^*, which we have then used its LSB to divide all the traces into two sets.

If the key guess is correct, we can imagine these two sets as collection of traces as drawn in the two figures above. In `set1` we will have traces looking like the left drawing and in `set0` we will have the traces looking like the left one.

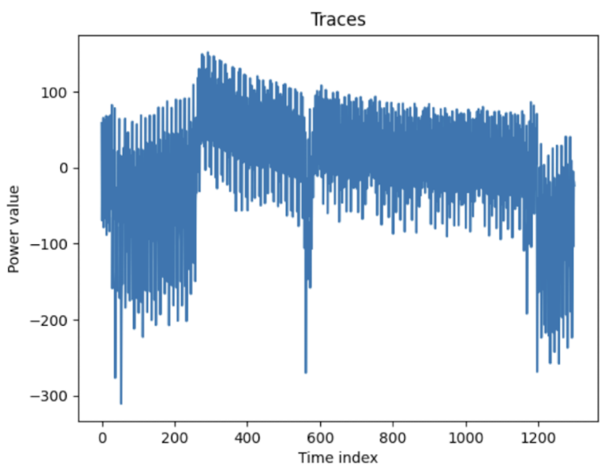
However, if the key guess is wrong, we basically divided all the traces randomly, so likely we will have as much traces as left and right ones mixed into two sets.

This hints that we can average them and maybe try to see how their averages look like.

```
        set0_avg = np.asarray(set0).mean(axis=0)
        set1_avg = np.asarray(set1).mean(axis=0)
```

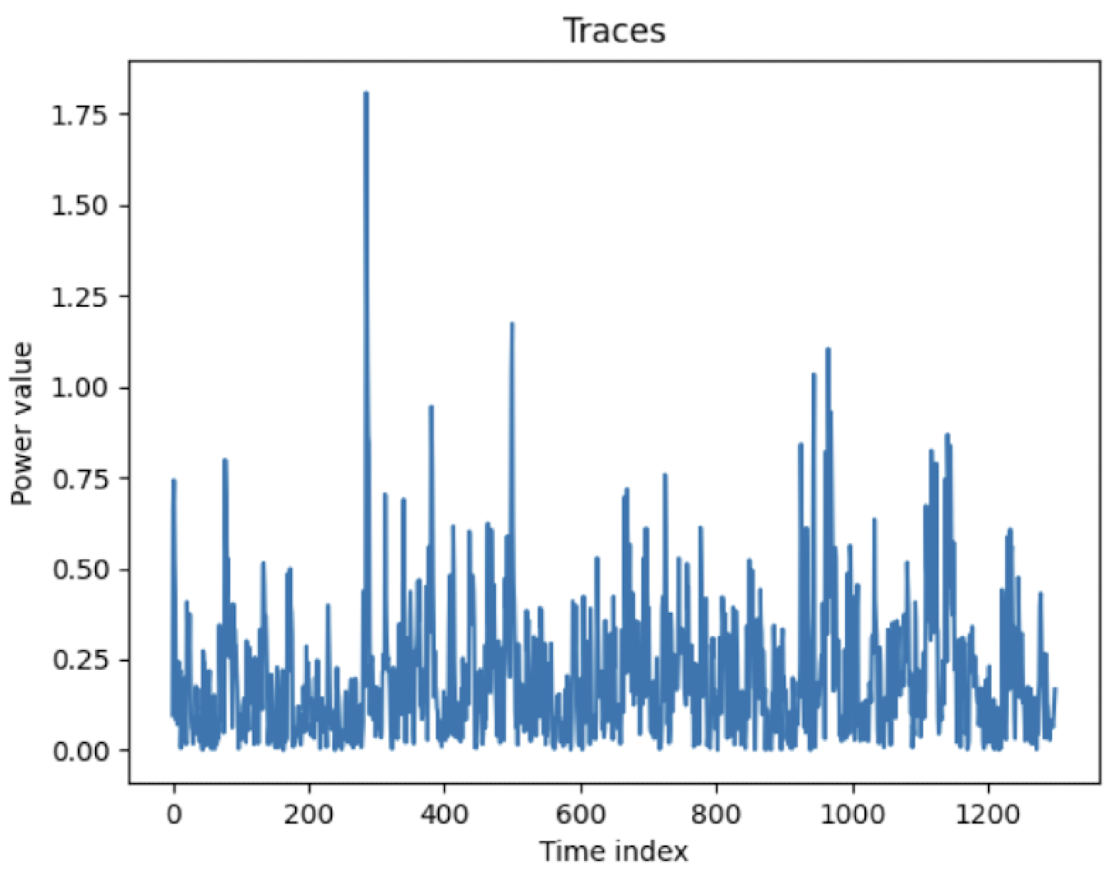


Set 0 avg



Set 1 avg

And let's see the absolute difference of means (absolute value of difference of means):



We are interested in the maximum value in this graph which is around 1.75. The larger this value is we can think that our key guess has some sort of relationship with the intermediate value being manipulated. This makes sense because if `set1` and `set0` are divided like I drew above, we should be able to observe a spike at the point where we have a difference and 0 at the rest. But of course there will be some noise present in the real measurements.

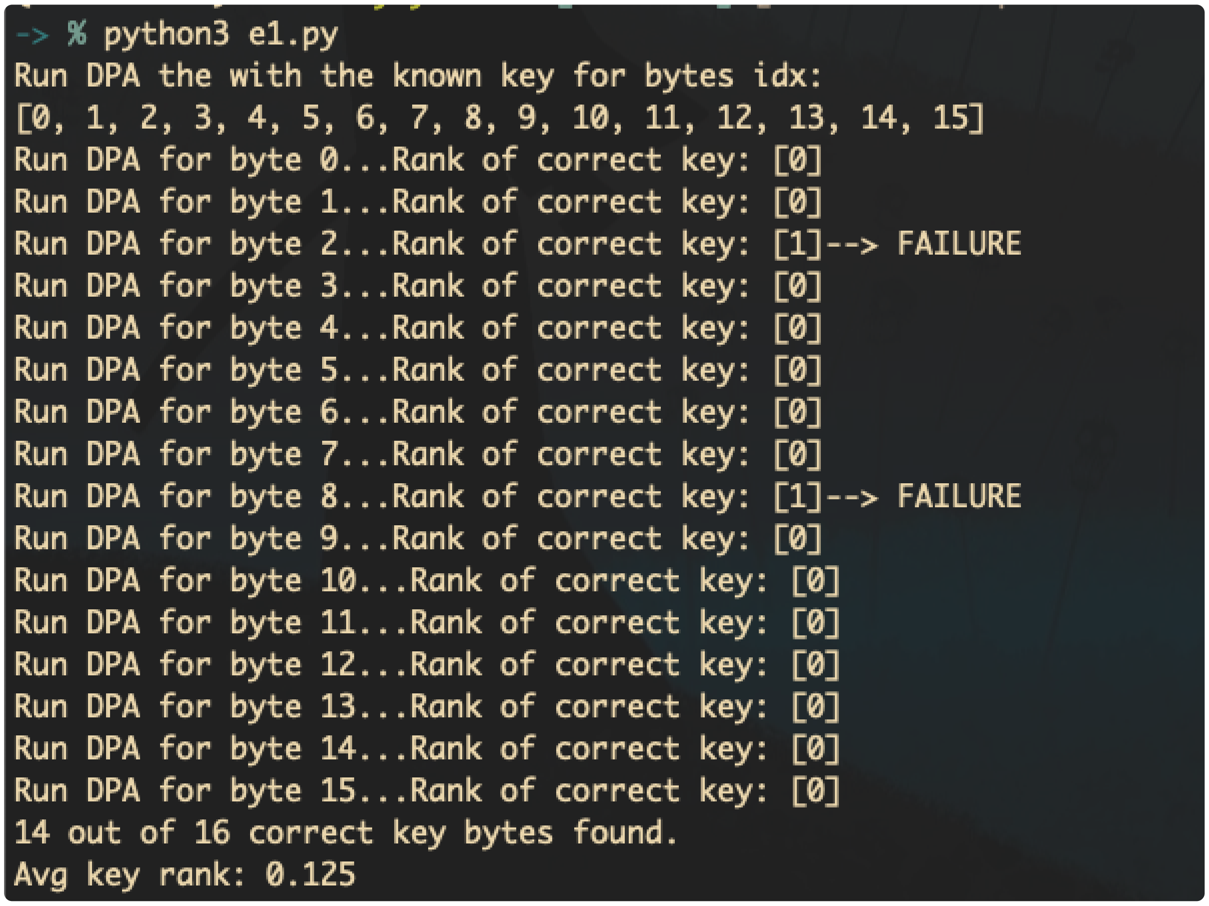
Our goal is to do this procedure for all the key candidates and then order key candidates based on this absolute difference of means. Then order the key candidates that give the biggest absolute differences of means to the lowest.

Complete code:

```
def dpa_byte(index, pts, traces):
    """
    index: index of the byte on which to perform the attack.
    pts: the plaintext of each encryption performed.
    traces: the power measurements performed for each encryption.
(n_traces, n_samples)
    return: an np.array with the key bytes, from highest probable to
less probable (256,)
    """
```

```python
    # TODO: Implement the function
    mean_diffs  = []
    for key_guess in range(256):
        set0 = []
        set1 = []
        for i in range(traces.shape[0]):
            x_i = pts[i,:][index] # get the byte of the plaintext
            v_i_star = sbox[x_i ^ key_guess] # apply the prediction
            trace = traces[i,:] # (16000,)
            if v_i_star % 2 == 1: # If LSB = 1 => modulo 2 returns 1
                set1.append(trace)
            else:
                set0.append(trace)
        set0_avg = np.asarray(set0).mean(axis=0)
        set1_avg = np.asarray(set1).mean(axis=0)
        mean_diffs.append(np.max(abs(set0_avg-set1_avg)))

    # sort i from 0 to 255 with key mean_diffs[i]
    return np.array(sorted(range(len(mean_diffs)), key=lambda i:
mean_diffs[i], reverse=True))
```



```
-> % python3 e1.py
Run DPA the with the known key for bytes idx:
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Run DPA for byte 0...Rank of correct key: [0]
Run DPA for byte 1...Rank of correct key: [0]
Run DPA for byte 2...Rank of correct key: [1]--> FAILURE
Run DPA for byte 3...Rank of correct key: [0]
Run DPA for byte 4...Rank of correct key: [0]
Run DPA for byte 5...Rank of correct key: [0]
Run DPA for byte 6...Rank of correct key: [0]
Run DPA for byte 7...Rank of correct key: [0]
Run DPA for byte 8...Rank of correct key: [1]--> FAILURE
Run DPA for byte 9...Rank of correct key: [0]
Run DPA for byte 10...Rank of correct key: [0]
Run DPA for byte 11...Rank of correct key: [0]
Run DPA for byte 12...Rank of correct key: [0]
Run DPA for byte 13...Rank of correct key: [0]
Run DPA for byte 14...Rank of correct key: [0]
Run DPA for byte 15...Rank of correct key: [0]
14 out of 16 correct key bytes found.
Avg key rank: 0.125
```

key enumeration and rank estimation
the adversary does the key enumeration

and rank estimation is done when you have the correct key
it is to asses how well the attack performs and then when you are actually attacking an implementation similar to what you have assessed your attack on you have some sort of an idea to how much you need to evaluate.