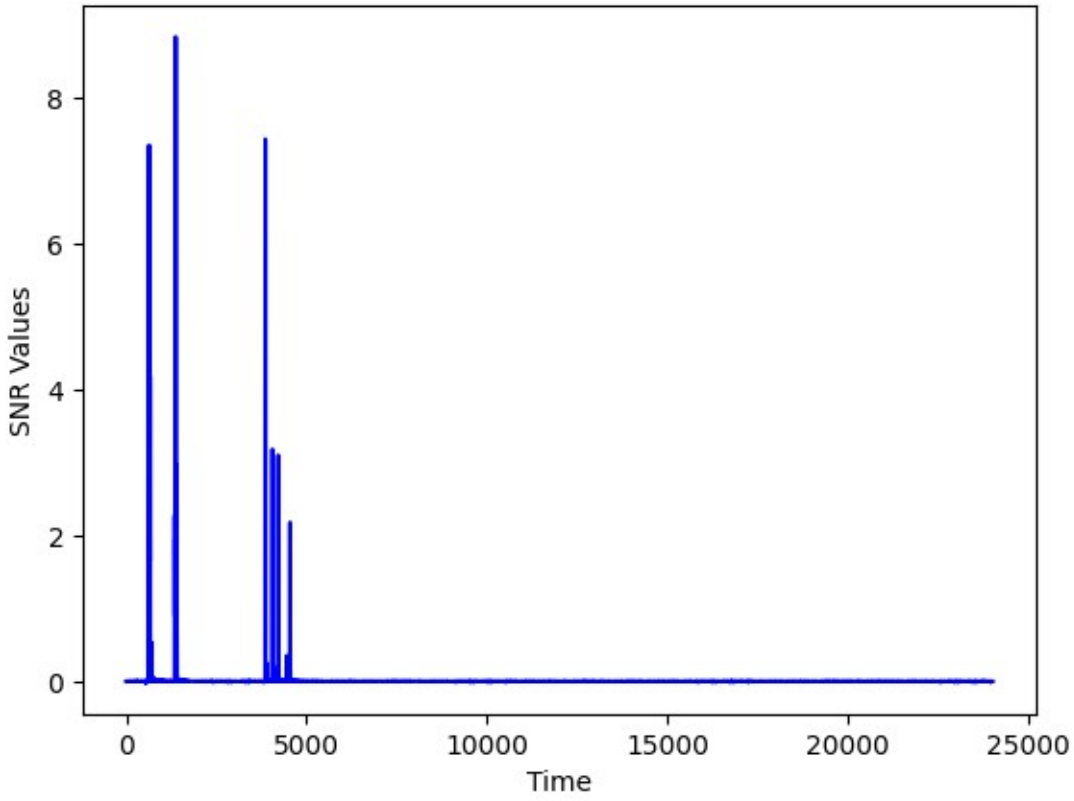
Note for Q:1 – The total points for parts (a), (b), and (c) (= 15) do not sum to the total points for Q:1 (= 20).

Q:1(a)
Device of type B is likely to show higher noise.

Q:1(b)
The device with the higher noise, that is, device of type B will be more difficult to attack.

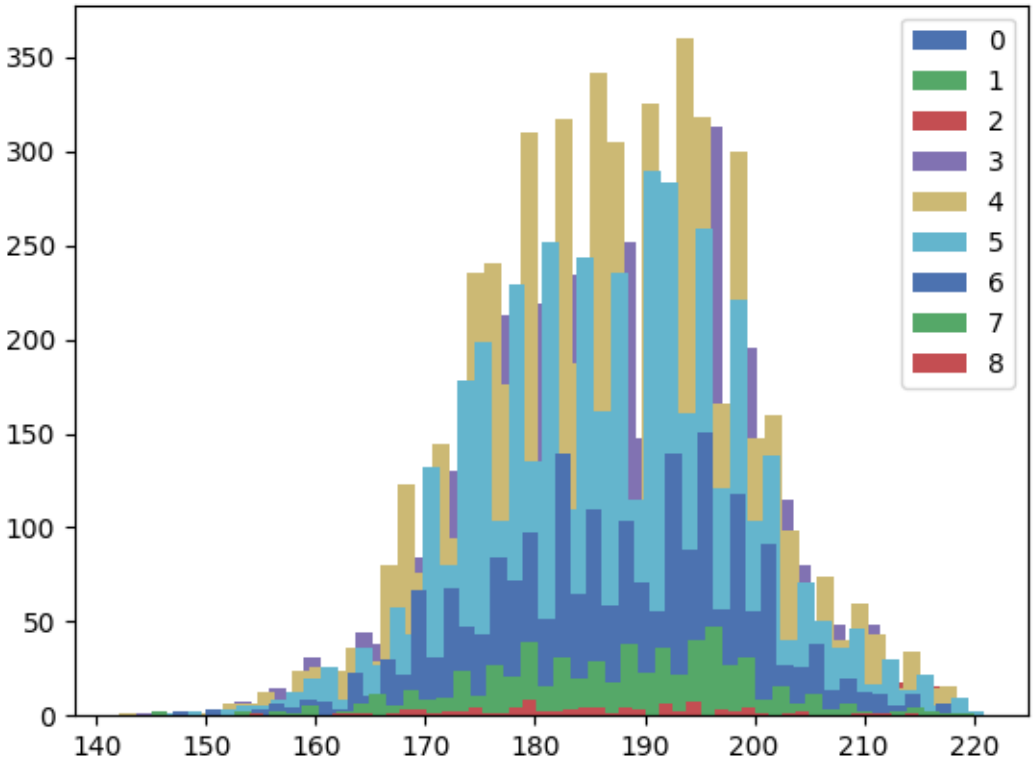
Q:1(c)
Yes. SNR (Signal-to-Noise Ratio) can be used for this purpose.

Q:2(a)
The SNR plot for the first keybyte is shown below:



The SNR plots for the other 15 key bytes can be found in the *snrplots* folder in the .zip file. The code used to generate this resides in the *snrplots.py* file.

Q:2(b)
The histogram plot for the highest SNR timepoint for first keybyte is:



The histograms for other SNRs with the highest timepoint are in the *histograms* folder in the .zip file. The code used to generate these histograms resides in the *makehistograms.py* file.

Q:2(c)

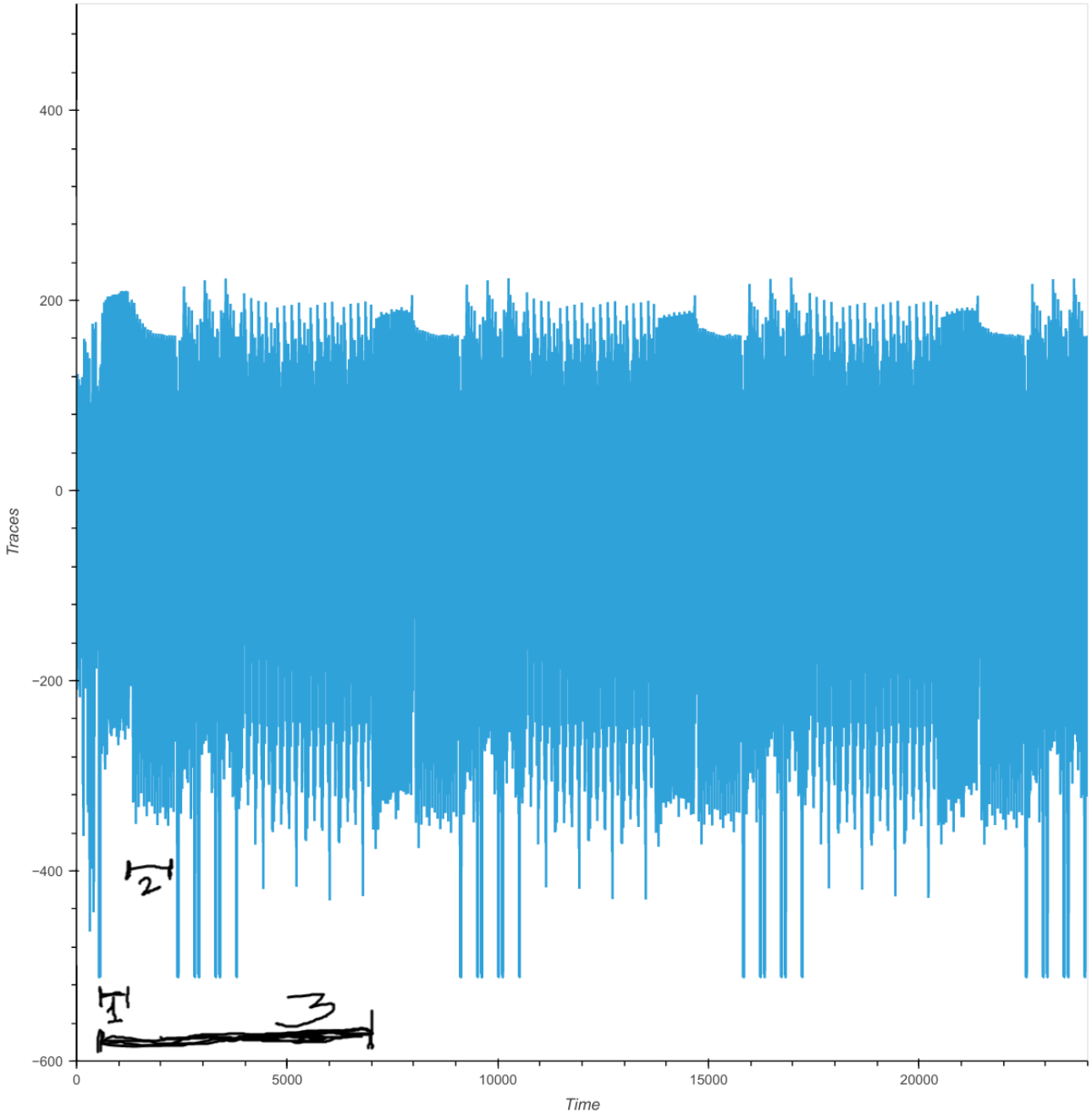
The horizontal line marked as “1” is where KeyAdd takes place.

The horizontal line marked as “2” is where SubBytes takes place.

And the whole AES round is marked as “3” by the horizontal line below it.

The code to generate this plot resides in the *annotateplot.py* file.

The annotated plot is: (this is for the first key byte, and plaintext = 0, as advised on Teams)



Q:2(d)
The key was recovered by running a CPA attack using the Hamming Weight leakage model on the SubBytes intermediate value. The recovered key was (decimal):
43 126 21 22 40 174 210 166 171 247 21 136 9 207 79 60

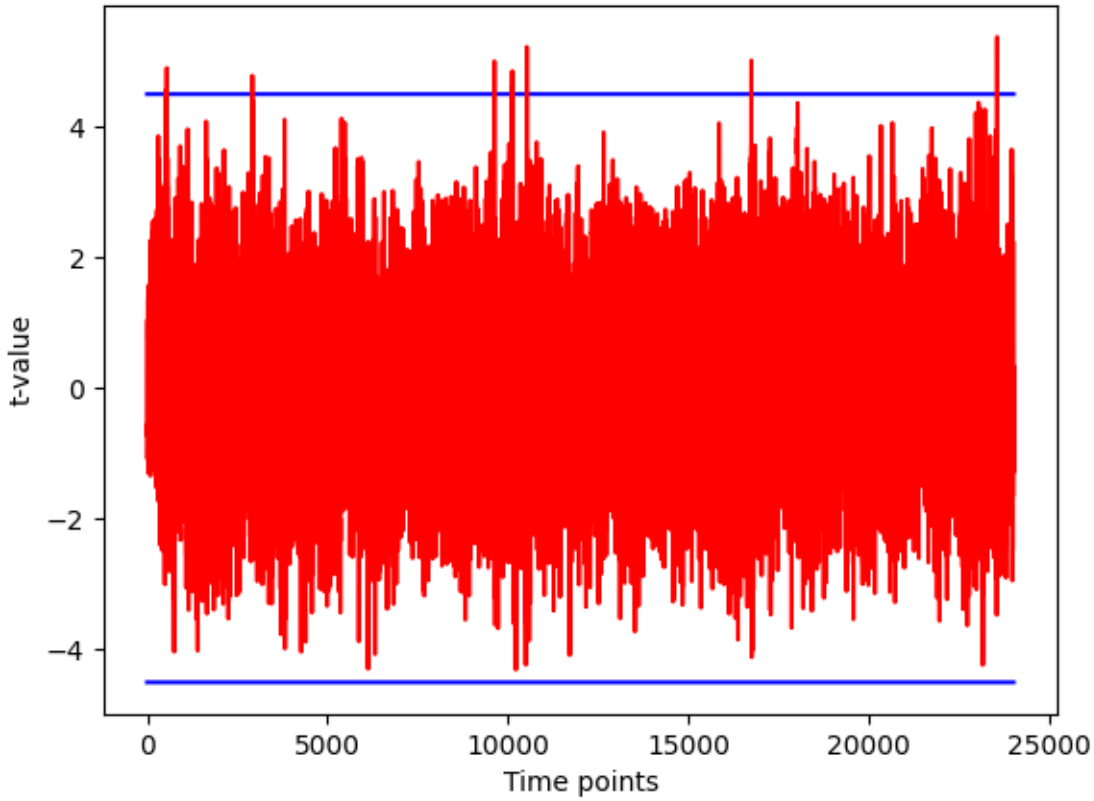
The Hamming Weight model on the KeyAdd intermediate value gives incorrect key for some byte positions.
The Hamming Distance model on the SubBytes intermediate value gives a completely incorrect key.

The code for the HW model on SubBytes intermediate value resides in the *cpa.py* file.
The code for the HW model on KeyAdd intermediate value resides in the *cpakeyadd.py* file.
The code for the HD model on the SubBytes intermediate value resides in the *cpahd.py* file.

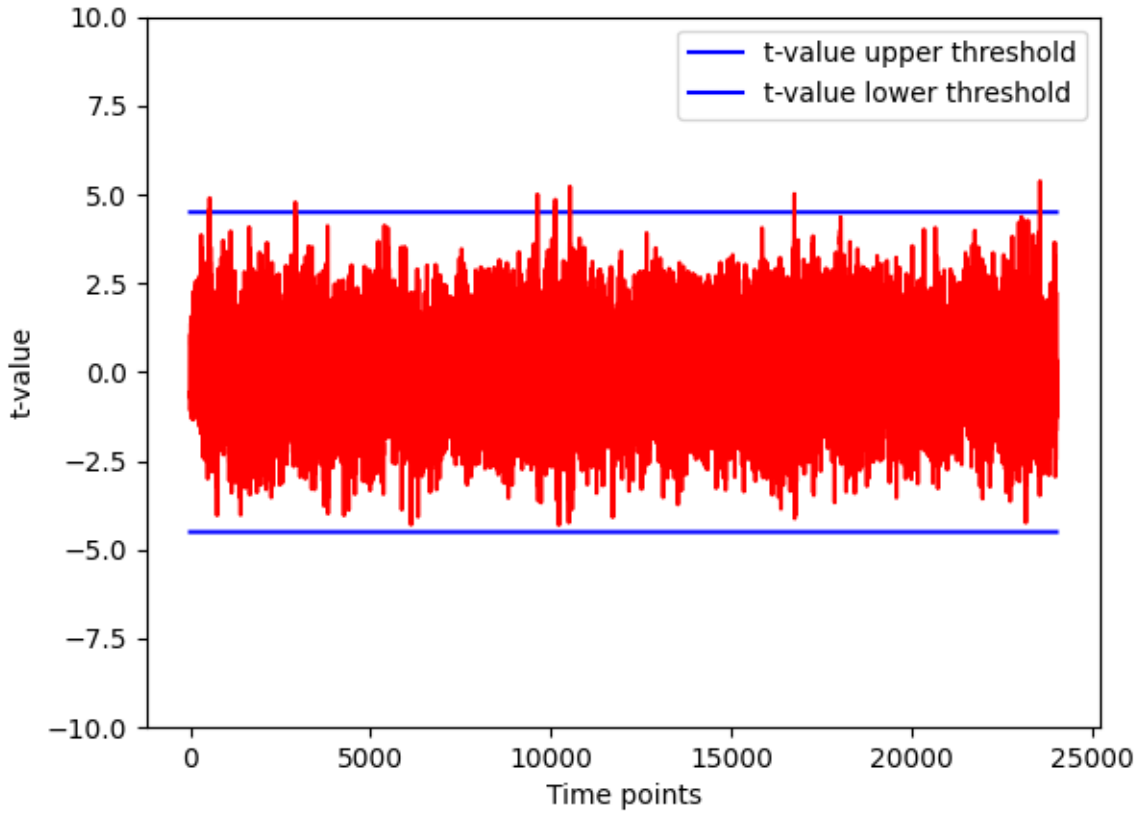
Q:3

I have implemented the Welch's t-test and the Chi-Squared test for this question.

For the t-test, the leakage plot is:



Another image, slightly more condensed for the same plot:

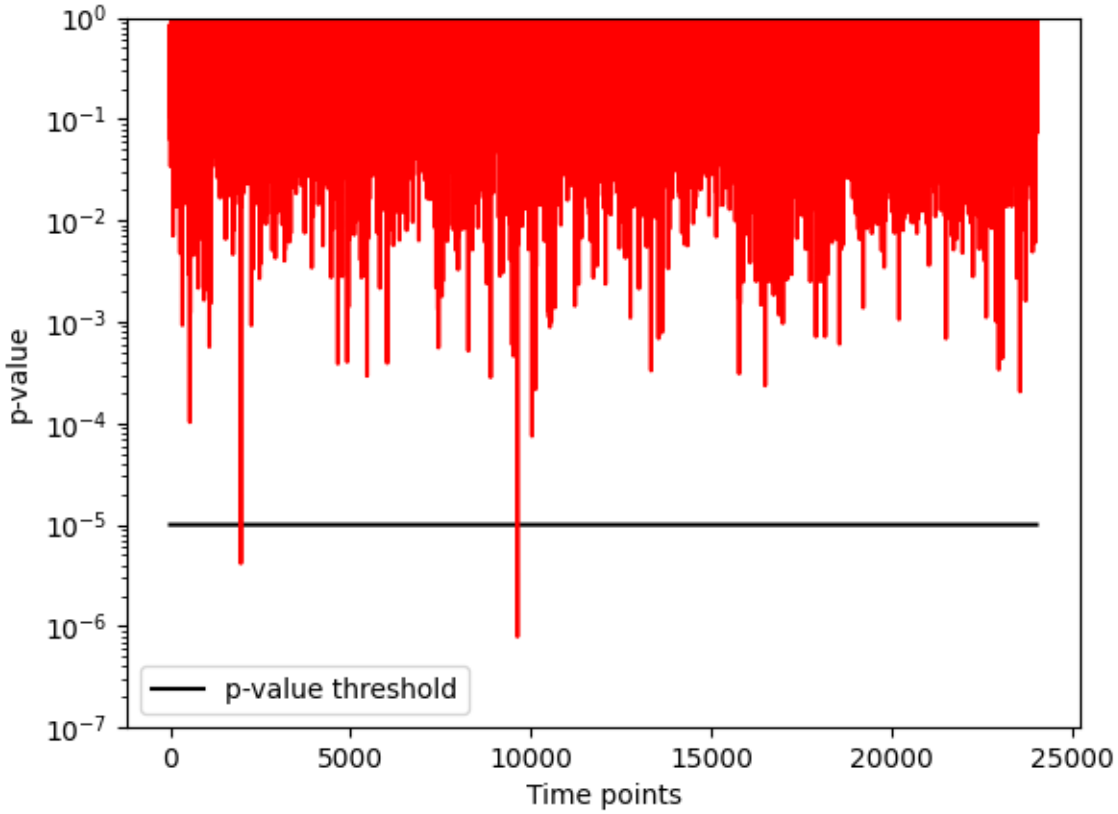


Thus, this implementation is vulnerable as leakage is clearly detected.

The t-test code ran for about 4 seconds for the first version (which I did using a loop), and for about 2 seconds for the second version which I did using Numpy everywhere.

The code for these 2 versions can be found in the *ttest1.py* and *ttest2.py* files.

For the chi-squared test, the detected leakage plot is:



Thus, leakage has been detected again.

It is worth noting that for 102 timepoints, the 2 groups of traces only had -512 as values, so it resulted in NaN values for the p-value array (because the value of degrees of freedom as well as chi-squared sum was = 0)

Also, regarding the code for the chi-squared test, it can be found in the *chisqrtest.py* file.

Some explanation for the code:

We first need to calculate the Fij table. This is easy: we combine the two tvlaarrays and extract the unique values and their frequencies (stored in the colcount variable).

Now, sum of values along each row will always be 10000 for i=0 as well as i=1. The sum of values along the columns we already have in the colcount variable.

Because the size of the 2 groups of traces are the same, the Eij table can be calculated simply by dividing the colcount variable by 2. And each cell along a column in the Eij table will have the same value because sum of rows is same, and size of the 2 groups of traces are same.

Thus, when calculating the chisqrd value of every cell = ([Fij – Eij]^2) / (Eij) , we need to consider an important fact: bins that are common in both the tvaluearrays and bins that are present in only one of these.

After solving the mathematical expression, it turns out that for unique bins, the value of their chi-squared sum is just Eij value itself. And we multiply it by 2, because as we explained earlier, the values along a column in the Eij table is same.

For bins that are common, it turns out after solving the mathematical expression, that we need to store a “delta” value in a different array for each such bin. This value is basically the frequency of that bin from either of the 2 groups of tvlaarrays. Then, the chi-squared value of that cell will simply be ([Deltavalue – Eij]^2) / (Eij) . For the bins that are unique, this deltavalue = 0, and hence the expression is simply Eij value itself, as stated above.

Once the chi-squared value of each cell is calculated, the rest of the code is straightforward.