Name: Hsuan-Chih Hsu

1. For this task, we consider two types of devices. i) device of type A implements the AES with just a single S-Box in hardware; ii) device of type B implements the AES with 16 S-Boxes to do a full AES round within just one clock cycle. Both devices are now studied as part of a power analysis attack. Let us consider the behavior of the noise in more detail:

a) In terms of a power analysis, which device is more likely to show higher (algorithmic) noise when assuming randomly distributed plaintext inputs?

Ans: Device B is more likely to show higher noise. Because device B has 16 Sbox, and this means it is likely to have the algorithmic noise to occur during the AES process, which also makes the attacker get in trouble of distinguishing the power consumption pattern of the AES process.

b) Assuming all other device characteristics are the same, which one will be more difficult to attack? Note that difficulty of attack is the number of traces required for a successful key extraction.

Ans: Device B. Because it has higher noise, which forces the attacker need more traces to recover the correct key. More clearly, the attacker needs more data to isolate the power consumption patterns that are related to the desired key.

c) Assuming there are no algorithmic countermeasures, is it possible to adapt the attack on device of type B such that when attacking the first round of AES, the attack would behave similar to device of type A? (hint: the attacker only controls the plaintext input)

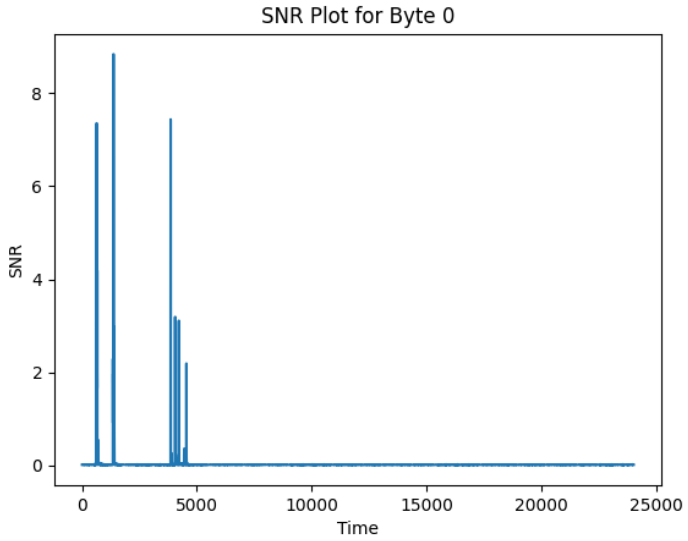
Ans: Yes, it is possible. Because the attacker can controls the plaintext input, and make only one Sbox is activated in device B, which would behave similar to the device of type A. More clearly, the attacker can choose plaintext that only one byte is effective in the first round, which would cause only one Sbox is used in the type B device.

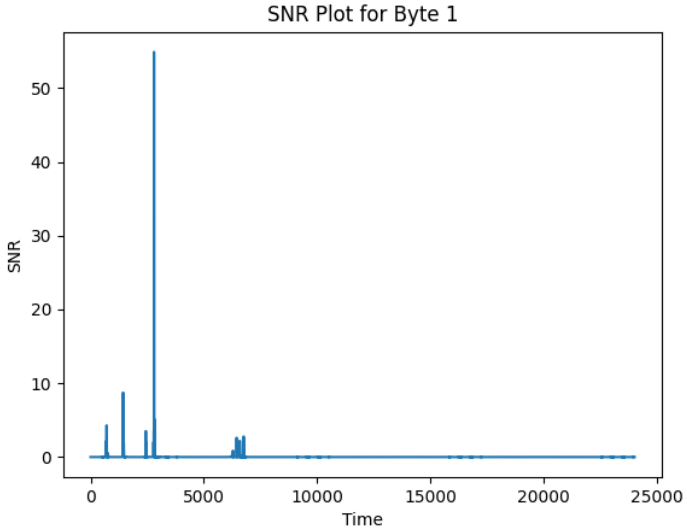
2.

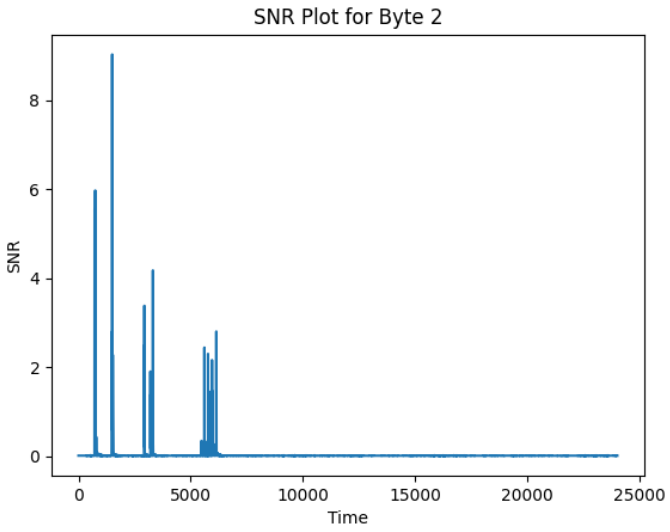
a) Create the SNR plot based on the plaintext input to quickly narrow down the number of points you have to work on.

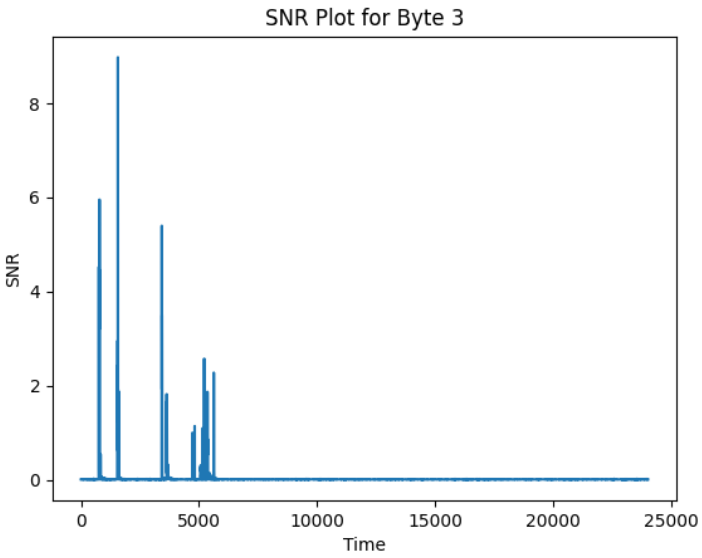
Ans: Code name: **task2_a.py**

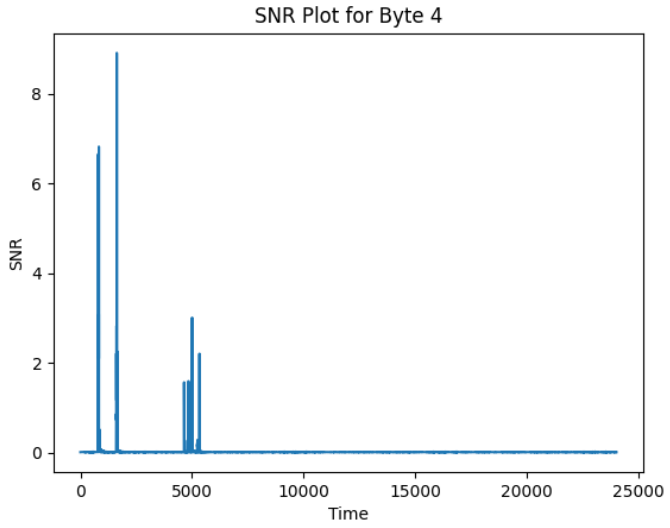
All 16 key byte figures are in the filename: **task2-a**.

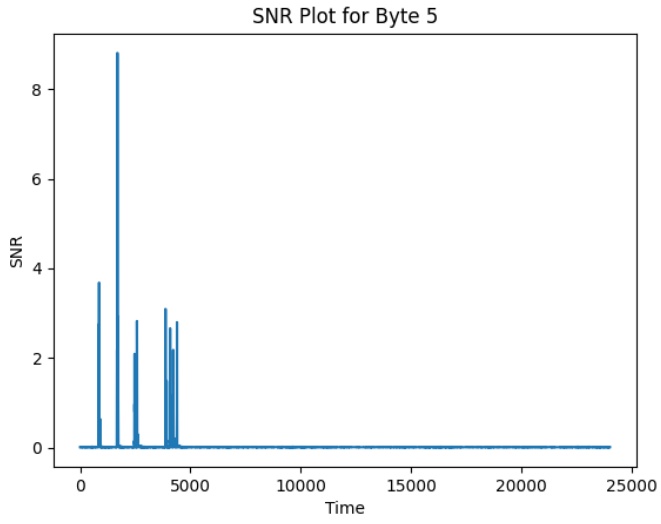


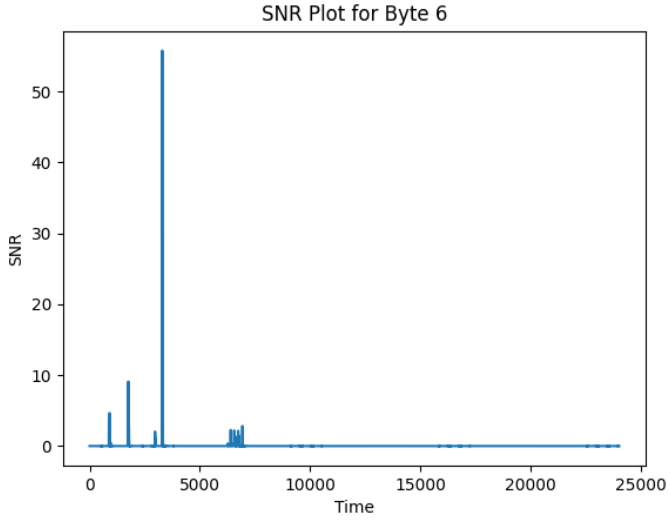


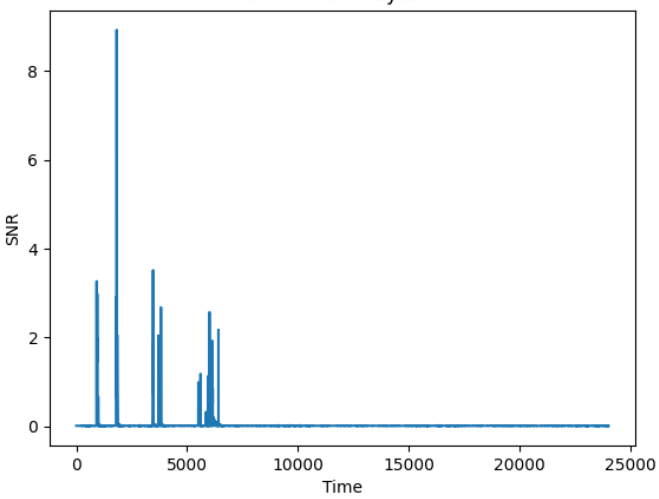


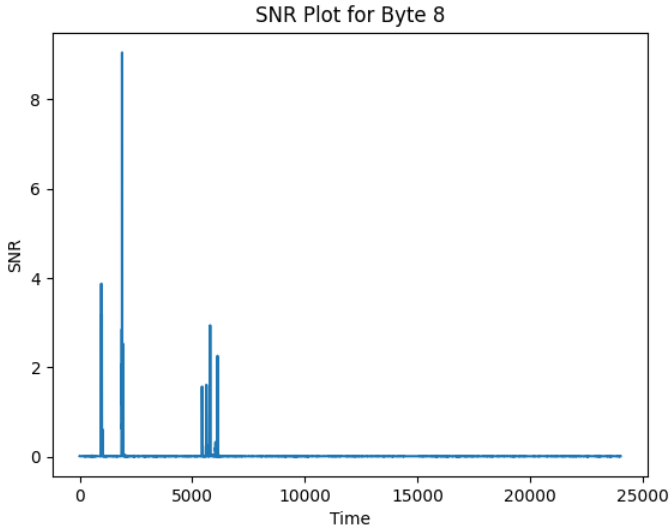


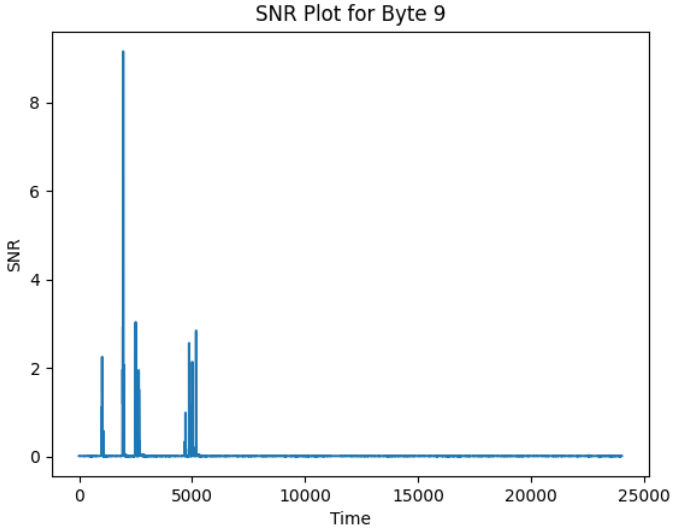


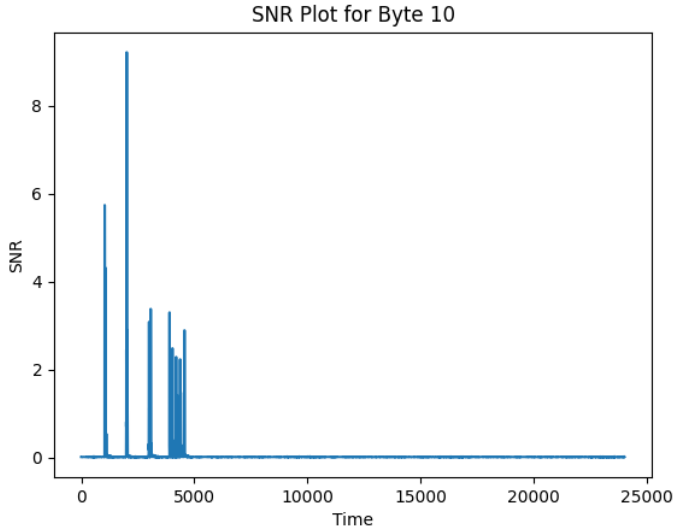


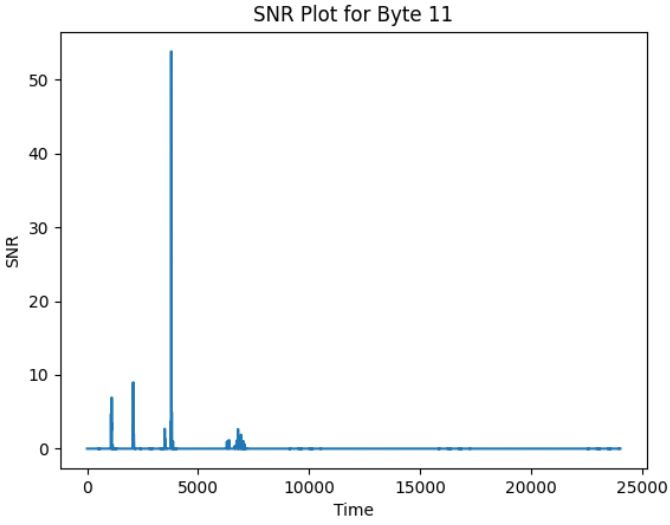

SNR Plot for Byte 6
SNR
Time
0
10
20
30
40
50
0
5000
10000
15000
20000
25000

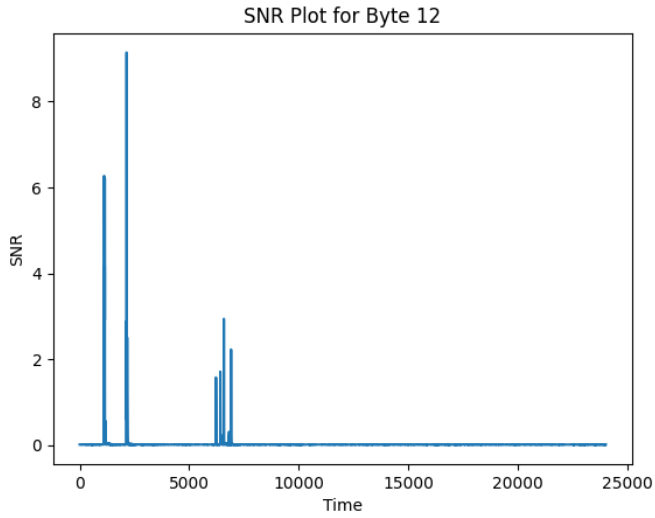

SNR Plot for Byte 7
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000

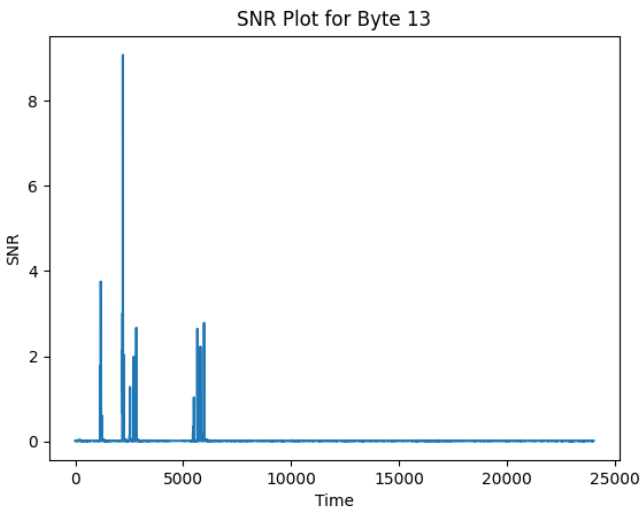

SNR Plot for Byte 8
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000

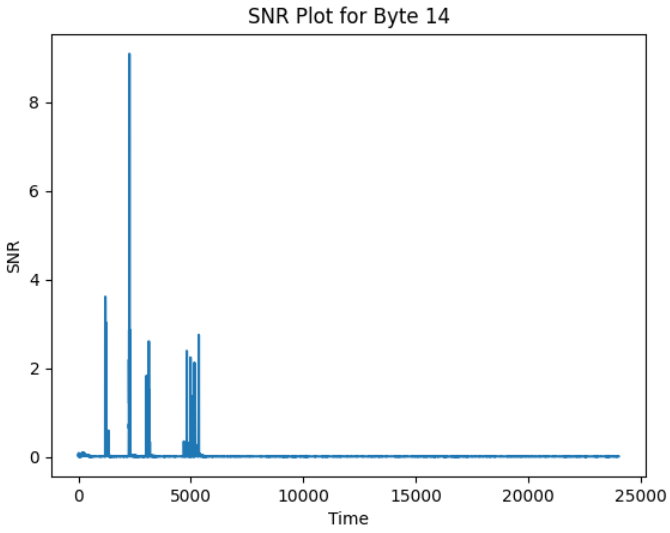

SNR Plot for Byte 9
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000

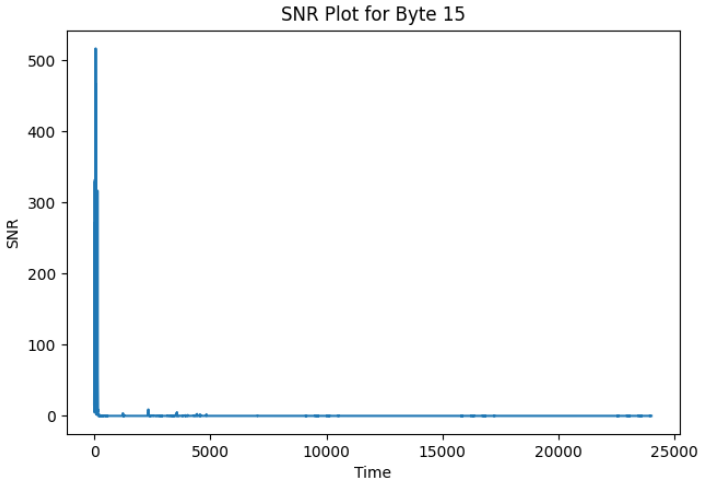

SNR Plot for Byte 10
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000


SNR Plot for Byte 11
SNR
Time
0
10
20
30
40
50
0
5000
10000
15000
20000
25000


SNR Plot for Byte 12
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000


SNR Plot for Byte 13
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000

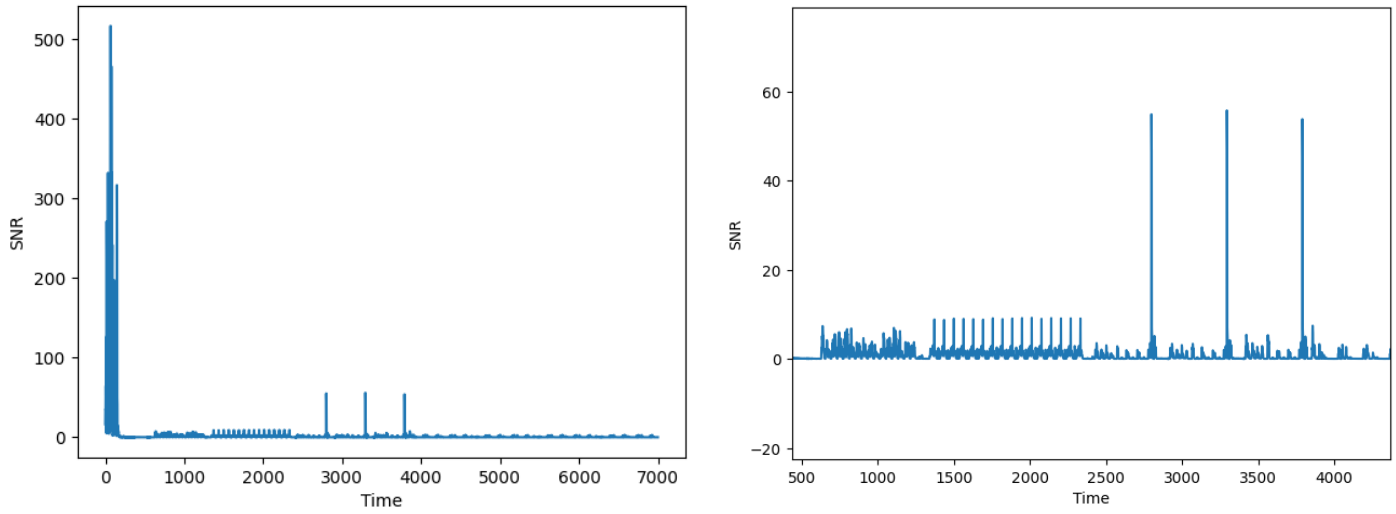

SNR Plot for Byte 14
SNR
Time
0
2
4
6
8
0
5000
10000
15000
20000
25000


SNR Plot for Byte 15
SNR
Time
0
100
200
300
400
500
0
5000
10000
15000
20000
25000

b) Create a plot where multiple traces with the same input data are averaged to remove the noise. Annotate the plot with at least: points in time where KeyAdd and SubBytes of the first round are performed. Identify the whole AES round.

Ans: The first (left) figure shows the whole AES round, but I think we must ignore the peak value of the beginning 500 timestamp. They are generated by the 15th key byte. The second (right) figure is much clearer. We can see the first group of 16 peaks which are the KeyAdd. And the next group of 16 peaks is the SubBytes.

Code name: task2_b.py
Time: 1851 seconds



c) Run a CPA to recover the key. Try different power models such as Hamming Distance/Weight, on different intermediate values such as KeyAdd and SubBytes. What do you observe? What works best and why?

Ans:
Filename: task2_c_hw_SubBytes.py
Key: [43, 126, 21, 22, 40, 174, 210, 166, 170, 247, 21, 136, 8, 207, 78, 61]
Time: 413 seconds

Hamming weight on SubBytes works better than hamming weight on KeyAdd. I believe this is because the leakage is more pronounced on SubBytes than on KeyAdd, which gives more secrets when we use the hamming weight on the SubBytes. Although I could still partially recover the correct key by using the hamming weight on KeyAdd, which is [43, 43, 21, 21, 40, 40, 45, 45, 84, 84, 21, 1, 9, 9, 79, 60]. But obviously, most of the key is wrong. File name of using hamming weight on KeyAdd: task2_c_hm_on_KeyAdd.py

3. Standard Welch’s t-test

Ans:

Code name: task3.py

Yes, the Welch’s t-test show the leakage at the time when the value is above the threshold 4.5. The time is listed as:

Time values where t-value > 4.5:

[535, 543, 2915, 9619, 10099, 10115, 10519, 16735, 23539]