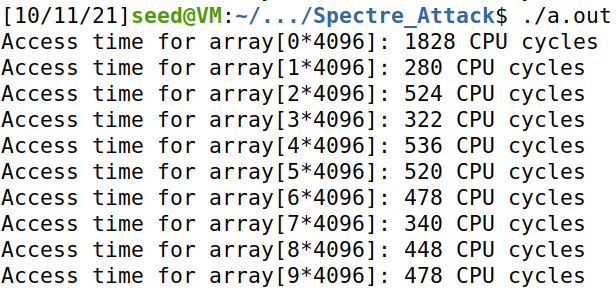
Homework 4

**Task 1**

For this task I compiled and ran CacheTime.c 10 times and measured the access time for each element in the array. The results are seen below in figure 1.



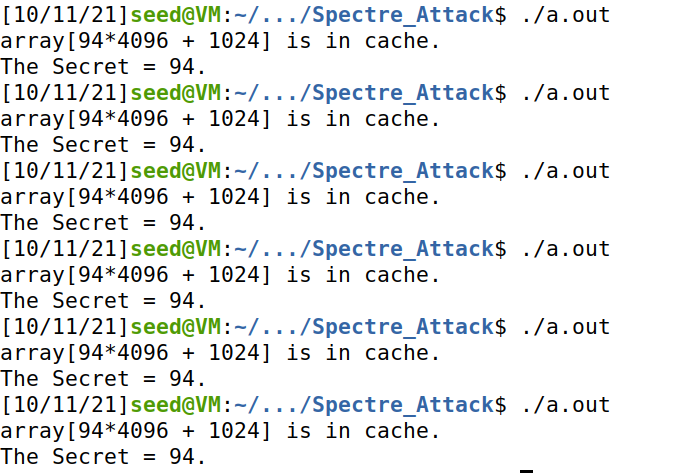
|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0\*4096 | 1\*4096 | 2\*4096 | 3\*4096 | 4\*4096 | 5\*4096 | 6\*4096 | 7\*4096 | 8\*4096 | 9\*4096 |
| 1 | 218 | 258 | 468 | 268 | 338 | 418 | 434 | 400 | 400 | 390 |
| 2 | 216 | 364 | 476 | 428 | 143 | 426 | 394 | 204 | 400 | 380 |
| 3 | 274 | 342 | 1256 | 404 | 414 | 434 | 428 | 340 | 390 | 1430 |
| 4 | 156 | 360 | 456 | 320 | 360 | 424 | 492 | 306 | 420 | 396 |
| 5 | 210 | 300 | 502 | 316 | 422 | 420 | 466 | 286 | 432 | 408 |
| 6 | 1934 | 314 | 502 | 346 | 510 | 514 | 520 | 506 | 560 | 500 |
| 7 | 184 | 282 | 536 | 324 | 564 | 520 | 556 | 408 | 516 | 500 |
| 8 | 1308 | 254 | 536 | 438 | 622 | 580 | 412 | 266 | 398 | 456 |
| 9 | 1924 | 322 | 574 | 370 | 530 | 540 | 500 | 356 | 480 | 526 |
| 10 | 1828 | 280 | 524 | 322 | 536 | 520 | 478 | 340 | 448 | 478 |
| A | 892.6667 | 307.6 | 583 | 353.6 | 443.9 | 479.6 | 468 | 341.2 | 444.4 | 546.4 |

**Figure 1:** a table of outputs in CPU cycles for 10 runs of the compiled CacheTiem.c file and an example image.

In the table for figure 1 there are 10 rows that show the 10 different runs of CacheTime.c. The A row at the bottom stands for the average of all the CPU cycles across the 10 runs for each element in the array. It can be seen in the table that some of the elements are able to be reached most faster then others. In this case the elements 3\*4096 and 7\*4096 are retrieved faster than most other elements in the array. My tests showed that the second element 1\*4096 was retrieved slightly faster but was no in the same range as 3 and 7. Because of how fast 3 and 7 are found on average a threshold can be made by comparing its average CPU cycles in comparison to the other elements. A larger threshold could mean that we can recognize when the CPU gets a hit more often, so looking at the averages acquired from the test the values that would conflict with us figuring out which of the elements are 3 and 7 would be elements 0, 1, 2, 4, 5, 6, 8, and 9 so I took the average CPU cycles of their averages and compared them to 3 and7. I got about a 520 average cycle for all the other elements and when comparing this to 3 and 7 this makes an almost 160 cycle difference. However, some of the elements other then 3 and 7 are much closer to their cycle value so it is important to see what the minimum cycle difference is between 3 and 7 and the other elements. The closest elements with similar cycles would be 1 and 7 which have a difference of only 34 cycles. The new threshold I believe should be somewhere in between these two differences. So, with that I came to a new threshold of (160+37) / 2 = 97. I believe this threshold will distinguish which elements are captured from the cache vs what is captured from main memory.

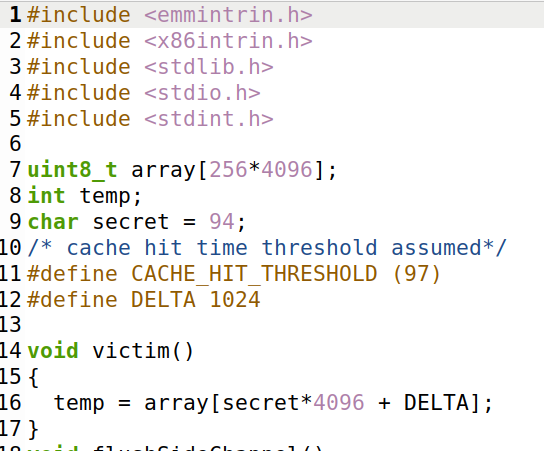
**Task 2**

For this task I compiled and ran the FlushReload.c file and ran the program 20 times. An example of the outputs of this program can be seen below in figure 2.



**Figure 2:** example image of it hitting the correct secret value from the array 6 time in a row.

In this task I was to count how many times the program was able to get the secret value from cache. After running it 20 times, I counted that the program hit 20 out of 20 times using the default CACHE\_HIT\_THRESHOLD value of 80. I changed this threshold value to the 97 value that I calculated in task 1 to see if it could find the secret values as many times as it initially did. This change to the code can be seen below in figure 3.

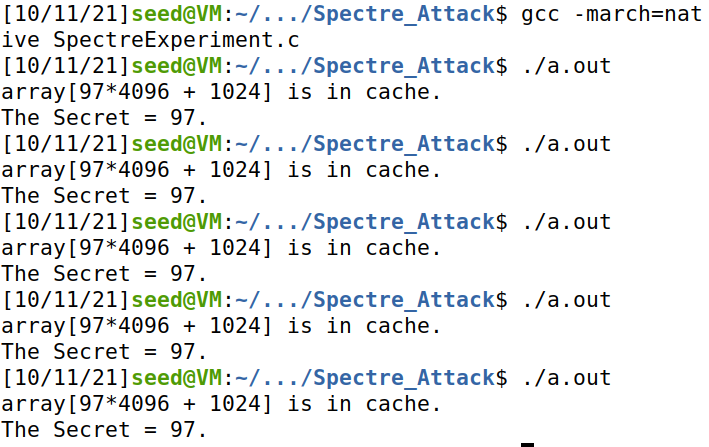


**Figure 3:** changing the threshold value to 97

It was able to hit 20 out of the 20 times I ran the code with the new threshold. I believe that this shows that both thresholds are relatively consistent at determining when an element is a cache hit.

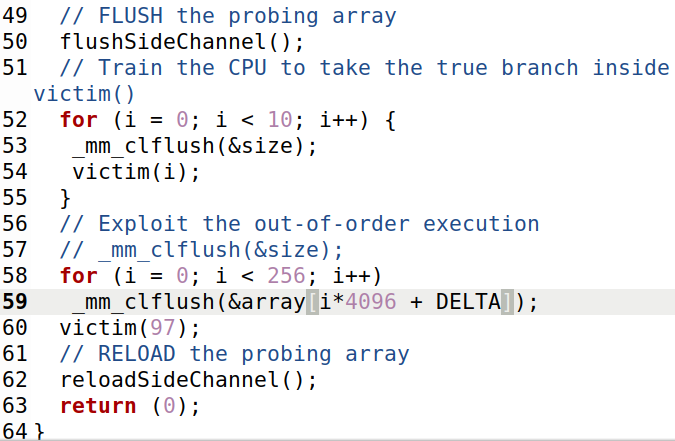
**Task 3**

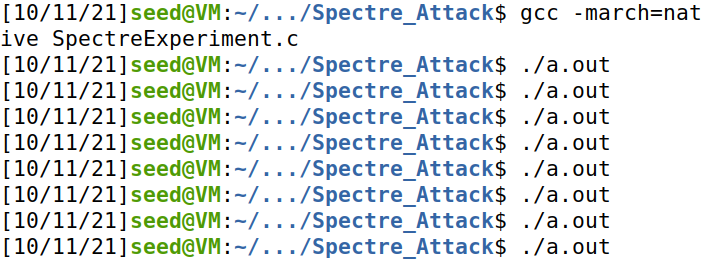
For task 3 I was to compile and run the SpectreExperiment.c program and explain my findings. The results of the first few attempts made can be seen below in figure 4.



**Figure 4:** The results of the first few tests of the SpectreExperiment.c program

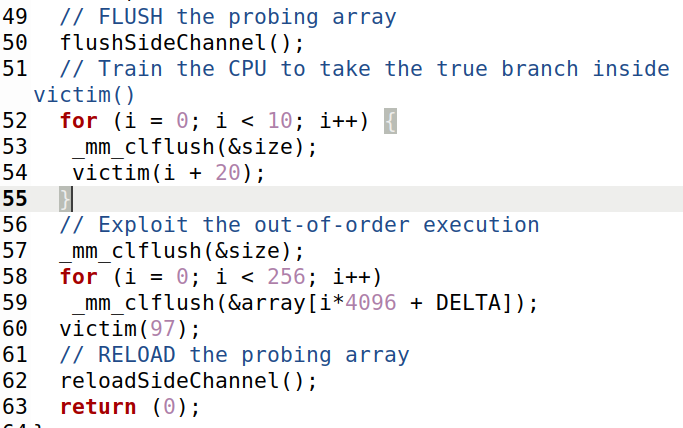
From the tests I did, I ran the program 20 times and got the secret value 20 times. From what the instructions said it seemed like there was supposed to be noise that would disrupt the CPU from getting the secret value every time, but it looked like from my test results that noise was minimal inside the channel. The task also wanted to see if the line 2 “temp = array[x \* 4096 + DELTA];” was ever run when 97 was passed into the victim function. This can be seen to be true since in the output examples I listed for figure 4 one of the outputs shows the printed “array[97 \* 4096 + 1024]” value. To further test this program, I made some changes to the code that can be seen below in figure 5 and 6. These changes included commenting out the call to flush out the cache before using out of order execution and changing the value passed into the victim function to be 20 larger then what was originally passed in, thus making the values passed into victim larger than the size variable and stopping the CPU from taking the branch to secret value.

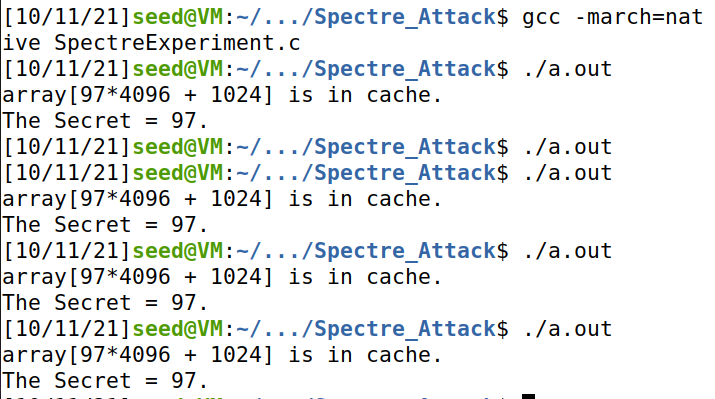




**Figure 5**: A change to the code that commented out the line that flushed the cache

After making this change the program was no longer capable of retrieving the secret value. I ran the program with this change 20 times and got no hits for the secret value, some of these results can be seen above in figure 5. This change to the code ultimately shows how clearing the cache is major step to successfully pulling off this attack. With out doing this there is too much noise, and it becomes virtually impossible to get the secret value.





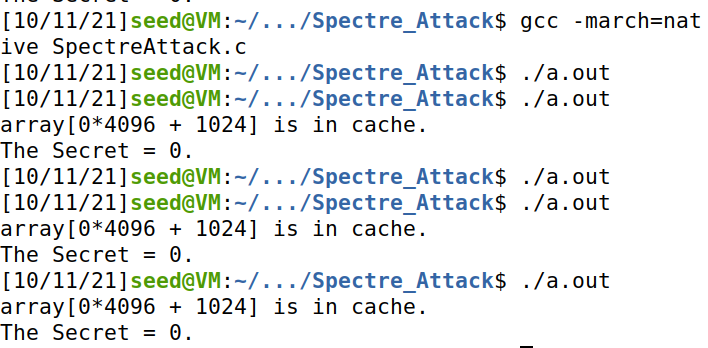


**Figure 6:** A change to the code where I added 20 to what ever was passed into the victim function

This change to the code caused the program to be much more inconsistent. By introducing a larger number into the victim function it was not able to acquire the secret value as often. In my test I ran the code 20 times and only hit 7 times. Four of these hits can be seen in the image above with a miss among them, the rest of the outputs resembled what was seen in figure 5. By changing the code, I made it so that the value being passed into the victim function was larger than the size variable, 10, which meant that the CPU was not loading the true branch to the secret value into cache. This made it much harder for the secret value to be measured using the threshold. In another test I did after the first one I noticed that changing the code in the exact same way led to some ability in getting the secret value to no ability at all in the code being able to get the secret value. This is likely because the secret value was never really loaded in the cache and so it was impossible for it to get the secret value using an out of order attack.

**Task 4**

For this task I had to compile and run the SpecreAttack.c program and observe its outputs to see if it was capable of getting the secret value. The results of some of this can be seen below in figure 7.



**Figure 7:** Some of the outputs of the ApecreAttack.c program

I ran the code 20 times and was able to get a hit on some value 17 times. This shows that there was some noise in the channel of the CPU that affect the program, but most importantly the code was not able to get the actual secret value. In the output of the program seen in figure 7 it has 0 as the secret value when the secret value is equal to "Some Secret Value". This shows that the attack was not getting even part of the actual value and needs improving.

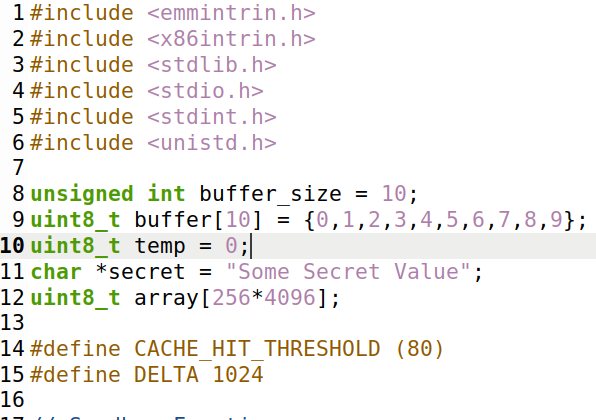
**Task 5**

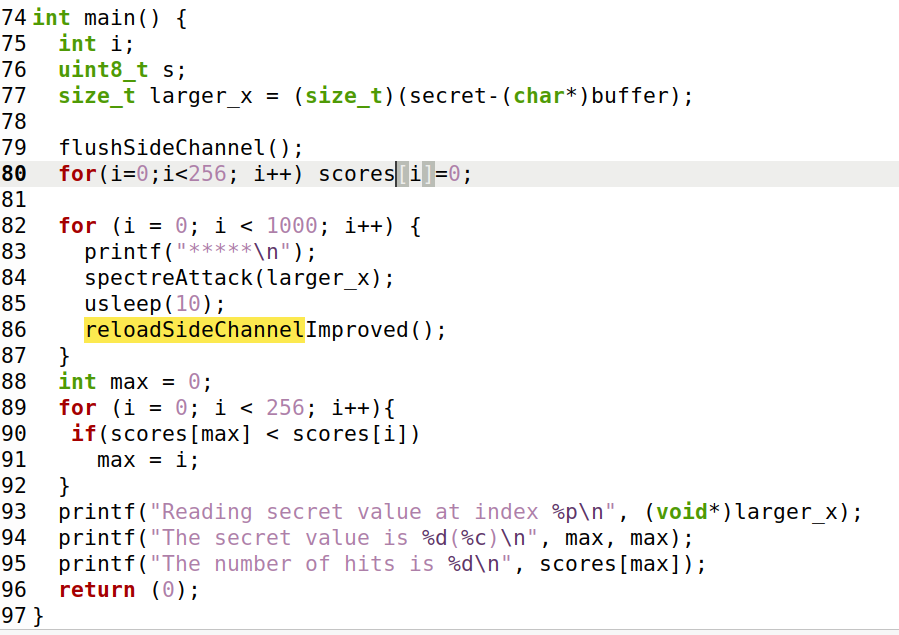
For this task I compiled and ran SpectreAttackImproved.c. The results of these tests can be seen below in figure 8.

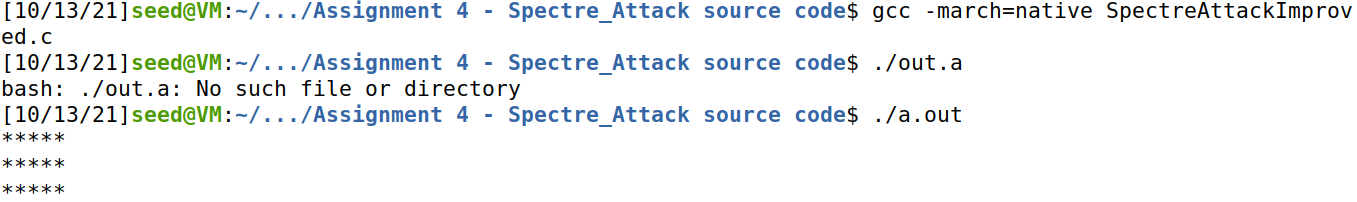


**Figure 8:** The outputs of SpectreAttackImproved.c

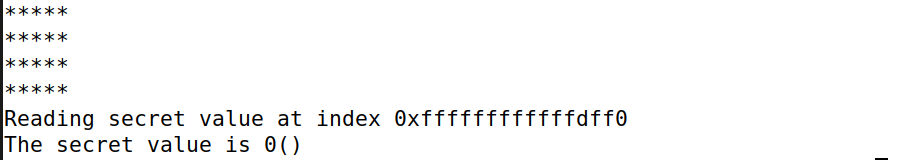
This initial test was run using the code that was provided in the zip folder found on BlackBoard. I edited this file to include the following to do the other requirements meant for this task. This changed code and its outputs to the same test performed above can be seen in the figure 9 below.



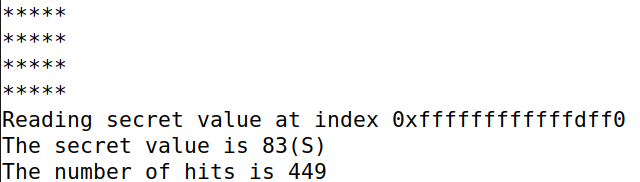




…

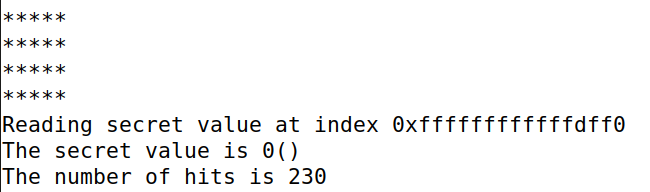


**Figure 9:** this figure shows the edits to that were made to the c file provided to make it applicable for the other tasks in the task. It also shows an example output of the program being run missing the secret value.

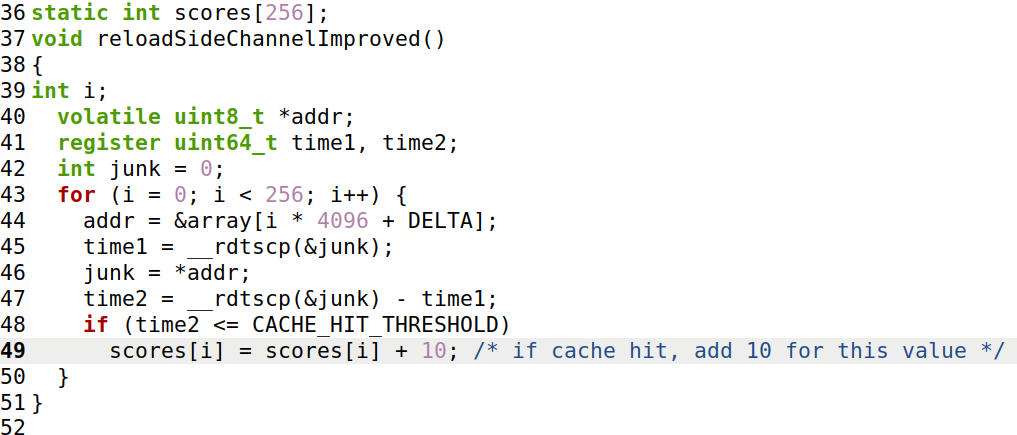


**Figure 10:** This is an example output of the program finding the secret value

From the tests that were performed on the edited c file I found that there was some noise in the channel of the cache and that it sometimes was not able to get any hits and could not find a secret value. This can be seen from figure 9 and 10 with 9 not finding a secret value and 10 finding one. However, there where also times when it did have hits and returned a wrong secret value 0. This can be seen in an output seen below.



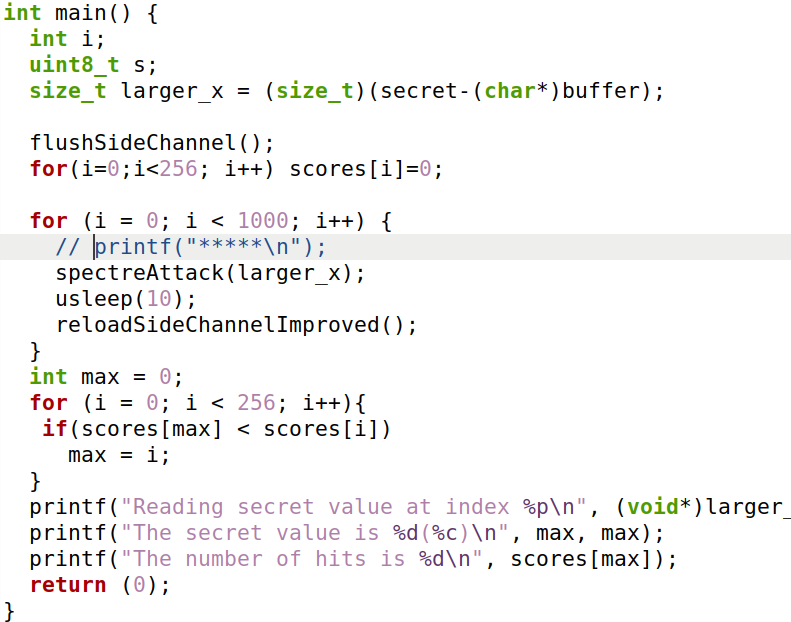
To fix this return of a false secret value 0 I edited the code to add more than 1 to the score[i] variable to make sure that max was set to the correct value. This can be seen in figure 11 below.

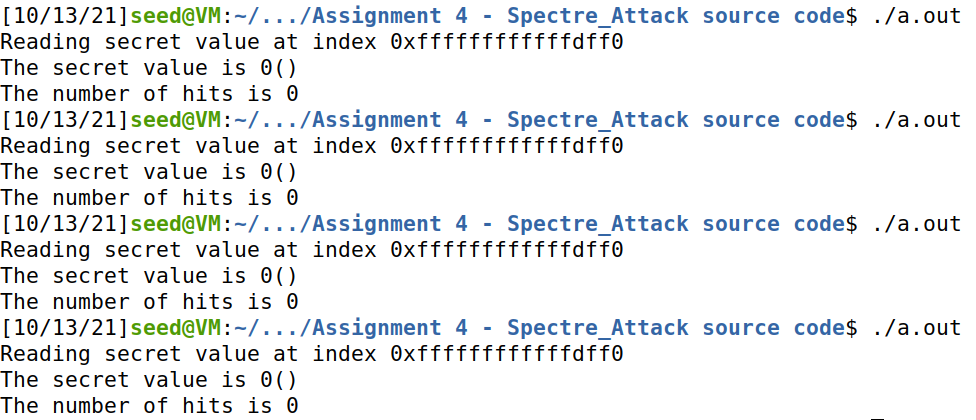


**Figure 11:** edit to the code to add more to the score so that it would more likely update the max variable

This change to the code seemed to stop the false secret values of 0 from appearing when the output said it had hits.

I was also asked to remove a line printf("\*\*\*\*\*\n"); from the code and observe the results. This can be seen below in figure 12.

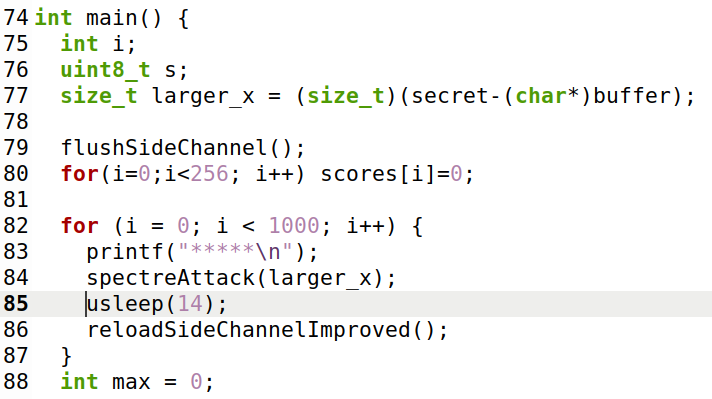




**Figure 12:** commenting out the printf line and the results of the program no longer being capable of finding the secret value.

With the line of code, it was printing 1000 lines of \* values every time it ran the specreAttack function. This was somehow making the spectre attack more successful and allowing it to actually get the secret value. Without it the program was no longer capable of getting the secret value. This could possibly be reducing noise somehow in the channel but I’m not positive how it would be doing that.

Lastly, I needed to test how the time spent by the program sleeping affected the accuracy of the program for finding the secret value. I did this by changing the value in the usleep call of the code from 10 to 6, 8, 10, 12, and 14 microseconds. I recorded these results in the table below seen in figure 13.



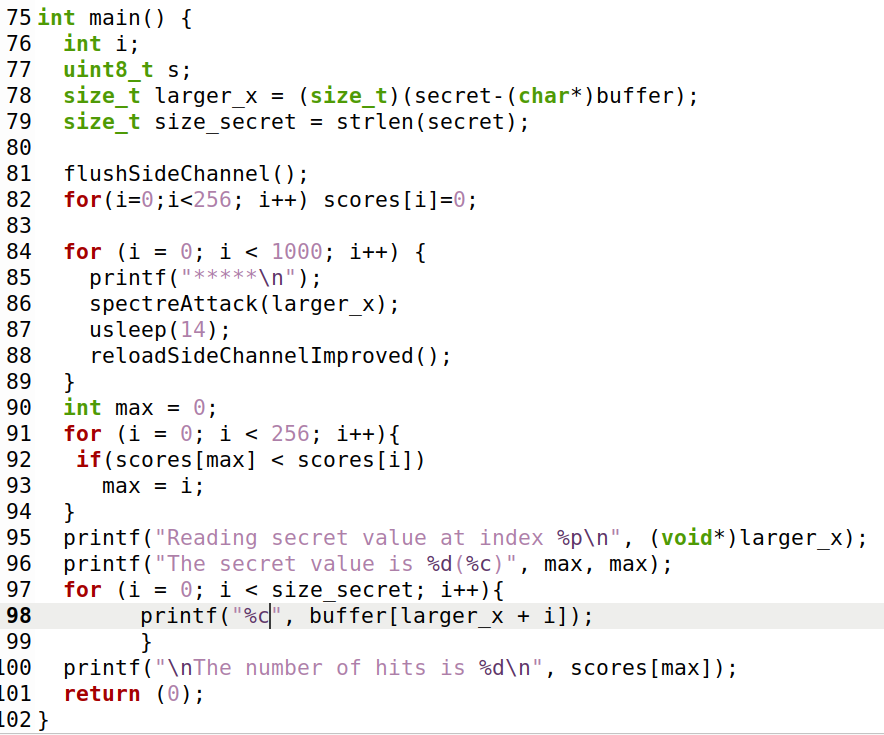
|  |  |
| --- | --- |
| Sleep score | Accuracy |
| 6 | 3/10 |
| 8 | 5/10 |
| 10 | 8/10 |
| 12 | 9/10 |
| 14 | 10/10 |

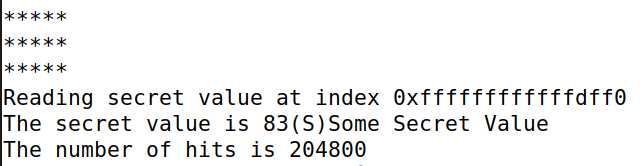
**Figure 13:** a table indicating the accuracy and the section of code that was changed with the different values seen in the table.

The sleep scores I chose were each put into the usleep part of the code and tested 10 times to get a rough accuracy on how much the change in time spent affected the program. I noticed that as the time spent sleeping was less the less accurate the program became and the fewer times it found the secret value. When the program had more time to sleep it performed better and got the secret value more often. This can be seen in the accuracy column of the table which shows how many times out of 10 tests the program returned a secret value with the given sleep score.

**Task 6**

For this task I needed to return the entire string of the secret value. To do this I made the following changes to the code seen below





This change to the code uses the size of the secret character array and the buffer to out put the characters in the secret array. I use a for loop that iterates for a number of times equal to the character array’s length and then prints out a character from the buffer starting from the location in the buffer where the secret string is located. This prints out the secret string in the output which can be seen directly after the code snippet above.