A side-channel attack is a security exploit that aims to gather information from or influence the program execution of a system by measuring or exploiting indirect effects of the system or its hardware -- rather than targeting the program or its code directly.

The Meltdown and Spectre attacks are side channels attacks and they made waves when they appeared because the vulnerabilities behind those attacks were built in the hardware and very hard to patch from the software (or patchable but with important loss of performance).

The two attacks we are talking about were using the time needed to access the memory as a side channel source of information. If you have an array of lets say 256 memory pages and you write only in one of them, if you measure the time needed to access each page you will find that the one in which you have just written will be accessed faster because was stored in the CPU cache. So, basically if you want to find out the value of a char and you have an array of 256 pages of memory and can access the n-th memory page without knowing n , you will be able to find n by measuring the time of access to each memory page and the index of the one which will be in the cache will be n.

Modern CPUs are executing instructions in advance to prepare the results before or while the access rights are checked, because of that memory which is not accessible to a process is accessed and although it is discarded and the process doesn't see it, still the effects like being in the cache remains.

Meltdown attack steps:

The CPU encounters an instruction accessing the value, A, at an address forbidden to the process by the virtual memory system and the privilege check. Because of speculative execution, the instruction is scheduled and dispatched to an execution unit. This execution unit then schedules both the privilege check and the memory access.

The CPU encounters an instruction accessing address Base+A, with Base chosen by the attacker. This instruction is also scheduled and dispatched to an execution unit. This way the attacker can be sure a memory was accessed from the range (s)he controls and can later check if it is in the cache or not.

The privilege check informs the execution unit that the address of the value, A, involved in the access is forbidden to the process (per the information stored by the virtual memory system), and thus the instruction should fail and subsequent instructions should have no effects. Because these instructions were speculatively executed, however, the data at Base+A may have been cached before the privilege check – and may not have been undone by the execution unit (or any other part of the CPU). If this is indeed the case, the mere act of caching constitutes a leak of information in and of itself. At this point, Meltdown intervenes.

The process executes a timing attack by executing instructions referencing memory operands directly. To be effective, the operands of these instructions must be at addresses which cover the possible address, Base+A, of the rejected instruction's operand. Because the data at the address referred to by the rejected instruction, Base+A, was cached nevertheless, an instruction referencing the same address directly will execute faster. The process can detect this timing difference and determine the address, Base+A, that was calculated for the rejected instruction – and thus determine the value A at the forbidden memory address.

Spectre attack:

Spectre is somewhat similar to meltdown in the way it take advantage by speculative execution , but in the case of Spectre it takes advantage of branch code preexecution. Lets take a look at this code:

```
int data_size = 10;

int array[] = {1,2,3,4,5,6,7,8,9,10};

if (x < data_size)

{

        char c = array[x];

}
```

The CPU if it is not able to preevaluate the value of truth for if (x < data_size) will execute in advance the instructions on the true branch, this way if the condition evaluates as true will be an important gain of performance and if not it will dump previous results. Problem is it will influence the state of the cache and this way this feature can be exploited to find secrets to which you would not have access in the first place. How to use the cache traces to read a secret will be explained below.

How to use cache to read a secret.

First of all lets see how we can invalidate cache from a CPU:

```
#include <x86intrin.h> /* for rdtscp and clflush */

static inline void clflush(volatile void *p) {

   asm volatile("clflush %0" : "+m" (*(volatile char *)p));

}
```

This function will invalidate the address p from the cache, meaning the value will be discarded from CPU cache and next time when it will be used will have to be re-read from main memory.

Practic. Create a program with an array of 1 million entries with values of 1. In a loop compute the sum of all elements 1024 times. Do the same but before accessing each element invalidate it from the cache. Compute the time to do that in both cases and tell if there is any difference.

Accessing memory is very fast but having the data in the CPU cache is even faster so to measure if some data was in main memory or in cache we need a very granular timer and the best one we can have is the one which measures the CPU cycles. To get the current number of CPU cycles elapsed you can use this function:

```
unsigned __int64 __rdtscp(
```

```
  unsigned int * AUX
);
```

Which returns the number of cycles elapsed and take one argument, an unsigned int where it will place whatever value is in register AUX, value which will not need to use.

Practic. Declare an array of char pointers like this:

```
char * array[256];
```

allocate for each element of this array a 4096 bytes buffer, like this:

```
array[i] = (char *) malloc(4096);
```

invalidate from cache each element like array[i][1000] where i=0…255

store a value in array[secret][1000], where secret is a value you choose.

Now measure the time in cpu cycles needed to access each element like array[i][1000], print it and see if you can guess from what you have printed the value of secret.

Ok, so at this point we should be able to guess a value of a secret based on the traces it left on the CPU cache. Now lets look at the following code:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <x86intrin.h>
#include <setjmp.h>
#include <signal.h>

sigjmp_buf _jmp;

struct gdt_desc
{
   unsigned short limit;
   unsigned int base;
} __attribute__((packed));
```

```
void store_gdt_desc(struct gdt_desc *location) {
  asm volatile ("sidt %0" : : "m"(*location) : "memory");
}

unsigned long inl_gdt(unsigned long gdt_mem_base)
{
   __asm__ __volatile__("sidt %0"
                  :"=m"(gdt_mem_base)
                  :
                  :"memory");

   return gdt_mem_base;
}
void sigsegv_handler()
{
        siglongjmp(_jmp, 1);
}



int main() {

        char * p = inl_gdt;

        signal(SIGSEGV, sigsegv_handler);

        if (sigsetjmp(_jmp, 1) == 0)
        {
                printf("first branch.\n");
```

```
        char kd = *p;
    }
    else
    {
        printf("SIGSEGV\n");
    }
    printf("Program still alive with stolen secret.\n");

}
```

What the code does – it registers a handler for sigsegv signal, it uses the functions sigsetjmp to set a jumping point meaning this functions returns 0 initially and saved the current execution flow to be able to restart the execution from that point. On the true branch where sigsetjmp returns 0 we are accessing a kernel address storing it in kd, this way we are trying to steal the value of an octet from kernel. Clearly this instructions will generate a segmentation fault , our handler will be called and in the handler we will call function siglongjmp which will make our program to jump and continue the execution from instruction sigsetjmp, but now sigsetjmp will return the value we just set with siglongjmp, this way our program survives our memory violation signal.

Practic. Now, on the branch which executes when the condition evaluates to true, hence sigsetjmp returns 0, imagine a mechanism in which you can influence the state of the cache to read that byte we read from kernel. If we can steal a byte from kernel, iteratively we can steal all the rest of the bytes, this way being able to access kernel structures, process memory and so on – not easy, but on a targeted attack worthing the time.

Unfortunately most probably the example from above will not work, the operating system in which you will try it being patched for such kind of attacks – in the future we will provide a virtual machine in a state in which it can be exploited.