Arjun Gupta

EECE5698

April 15th 2019

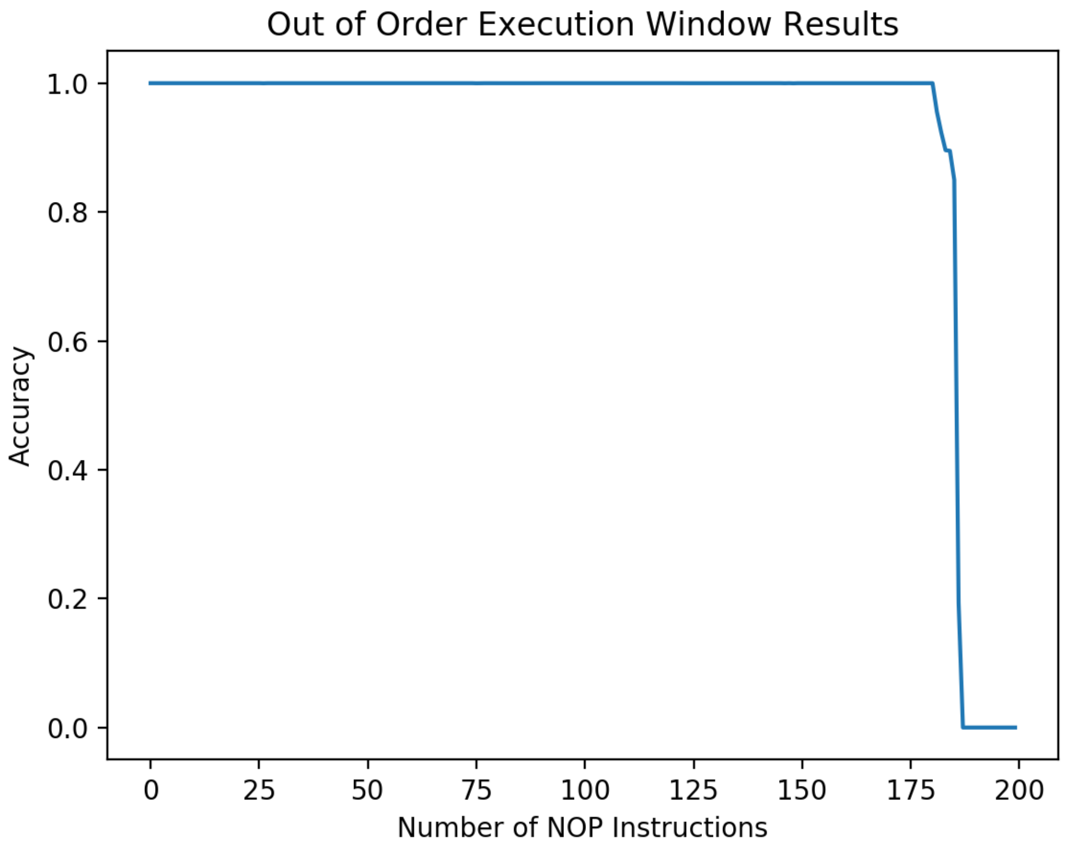
Lab 6 Report

In order to properly calculate the Flush+Reload threshold, I employed a tactic that was seen in the original Meltdown source code. Firstly, I allocated a space of 10 bytes and measured both cache hit access time using the "reload" method of first accessing the memory space in order to cache the memory space, and then consistently "reload" the memory into a cache line by timing how long a memory access takes. Then, I performed Flush + Reload by running clflush on the address in order to flush the cache line that the specific address is contained in, and then timing how long reloading the memory into the cache takes. With the ECEHSS server configuration, the cache-hit time was around 38 CPU cycles, while the Flush + Reload time was around 323 cycles. The threshold calculated by taking 80% of the cache-hit time plus 20% of the Flush +Reload time to obtain: (0.8*38) + (0.2*323) = 95 CPU cycles. This threshold was used in calculating the reliability of the covert channel.

In order to calculate the reliability of the covert channel, a userspace array of 256 bytes * 4096 bytes was constructed, where each array index was spaced 4096 bytes away from eachother in order to prevent cache prefetching. Next, a random byte (0 to 255) was generated and encoded into the cache by accessing that value in memory using the move quadword, *movq*, assembly function. This verified that the random byte was encoded into an index of the user array. Next, the userspace array was iterated by reloading the cache until a timing value was found that was below the given threshold, and that value was returned. This process was done 100000 times, with 88366 correct values, giving the covert channel a 88% correctness value.

On the ECEHSS server, it is running a quad-core Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz. This was discovered by printing out the status of /proc/cpuinfo.

Next, the OOO execution window was tested. This was done by modifying the same covert channel setup code to access kernel address values, which would trigger a fault. This was done by creating a signal handler that would respond to SIGSEGV, or segmentation faults, and unblock the signal. This would also jump past the faulty access in order to continue the execution loop. In order to measure the execution window, a "NOP" sled was constructed to barrage NOPs into the instruction order. The amount of NOPs ran from 1 to 200, and as the amount of NOPs increased, the reliability changed. A value of 10 was encoded into the cache and copied over as an index into a user array. This was all done in the out-of-order process. The figure below demonstrates the reliability from 1 to 200 NOPs in a row.



Next, the full Meltdown attack occurred. This was done by running the same code done at a specific target address where the program was executed in kernel space. An infinite loop was used to continuously read out bytes from the privileged program. The output was as follows: “This is the secret: Happy Hacking!”.

In order to prevent Meltdown, a patch to the Linux Kernel that added Kernel Page Table Isolation (KPTI) was added. This would separate page tables into kernel page tables and userspace page tables, allowing for minimal system calls and hiding other kernel-level functions in user space mode. Secondly, a change in CPU architecture to provide extra security for OOO execution, or removing OOO execution altogether would help to mitigate Meltdown.