Practical Implementation and Security Analysis of Meltdown on x86 Based Virtual Machines

**This color means that task is complete**

1. Intro – flush + reload, spectre, meltdown
2. Motivation – security implications, practical implementation
3. Methodology – **measuring cache time (struct timepsec, timeval, rdtscp)**, flush + reload – thresholding vs min.time approach, Kernel Modules, vmalloc and kmalloc, encouraging data to remain in cache (procfs, prefetch, etc.), trapping exceptions in C, delaying commit – keeping ALU busy.
4. Results
5. Mitigation – KPTI, microcode patches, linux kernel versions
6. Mitigation Analysis
7. Conclusion

**Methodology**

We start by stealing values local to the process using cache side channel attack. The possible methods for timing memory accesses are explored and optimal timing technique is arrived at. We then look at implementing flush reload and check its efficacy.

**Measuring Memory Access Time**

We need very high level of accuracy when measuring time taken for memory access. The general paradigm for this involves reading CPU wall times before and after the access and measuring the difference between them. There is a large variety of APIs for getting the CPU wall time. For our purposes, we require an API that satisfies the following criteria.

1. Measurement is sufficiently precise, i.e., captures adequate granularity of time
2. The measurement is highly accurate. This implies that there should be minimum delay between the issue of the call and the return of wall time by the CPU. This involves taking care of various overheads, such as context switching to the kernel space, delay between CPU issue of clock read instruction and the return of clock value.

Calculation of wall clock time, i.e., the actual time is very expensive, as it requires conversion of counter values into wall clock format and synchronization of the CPU with known accurate times. A key observation here is that the wall clock time is not required, as we are only interested in finding the difference between time before and after the access.

The clock counters available in Linux are:

1. **HPET: High Precision Event Timer**. Jointly developed by Microsoft and Intel and incorporated in chipsets since 2005. It gives a precision of around 100ns.
2. **Acpi\_pm clock:** This clock source is obtained from the ACPI Power Management Timer. The advantage of this clock is the frequency is not dependent on the power, hence is unaffected by power changes. It tuns at 3.85Mhz, hence gives a precision of 280ns. It is more expensive to query and less accurate than the HPET.
3. **TSC: Timestamp counter.** This counter is started during bootup and is driven by the CPU crystal oscillator. Hence it operates at the same frequency as the CPU. This enables it to run at extremely high update frequency. For CPU speed of 3Ghz, it is updated 3 times in a nanosecond. This is the most accurate clock source. It is easy to access as it just involves reading a register value.   
   In older CPUs it is important to fix the frequency of the timer as the frequency of the CPU is not constant. This limitation is removed in modern CPUs by having invariant TSC, i.e., the TSC is run at a fixed rate.

Hence, we choose the TSC as the clock source for maximum accuracy and high speed of response.

**!!IMAGE : $cat /proc/cpuinfo | grep -i tsc**

We make use of the RDTSCP C API for reading the contents of the TSC. RDTSCP is used to prevent out of order execution from giving incorrect time values. It performs any necessary serialization itself and is more efficient than manually enforcing serialization in the code.

**Flush + Reload**

We create an oracle buffer of 256 entries, each entry is of size 4KiB. This is done because the page size is 4KiB and may cause multiple entries to be cached as a result of cache locality.

The first step is the flush operation. All the entries of the oracle are initialized and then flushed from the cache. This ensures that none of the oracle entries are cached beforehand, and prevents incorrect results.

Second step involves the execution of victim code. The victim uses a secret character (local to the victim function) to modify an entry in the oracle. When this happens the oracle entry gets cached. No further operations are performed by the victim to avoid overwriting the cache.

The third step is the reload operation. The time taken to access each entry in the oracle is measured. The entry that has minimum access time is the one cached by the victim. Hence, we have obtained the index of the cached entry which is can then be used to get the secret value.

Minimum access time on Oracle[k]

Victim caches oracle[k]

0 1 2 3 ……………..... **k** ……… 256

**Oracle Buffer**

**Cache**

Time access to each element of oracle

**Attacker**

Another method of performing flush+reload involves determining a threshold value for cache access time. In this way, we take the first access whose time is less than the threshold as the cached value. This offers some reduction in computation to be performed.

We found that both techniques yielded similar success rates, and we choose the minimum time technique since the amount of computation is not high, and it gives a better guarantee of correctness.

**!!IMAGE: FLUSH + RELOAD using both techniques**