Benjamin Vargas

May 15, 2018

Invisible Bugs: The Spectre Vulnerability

Spectre is a sophisticated computer security vulnerability that preys on memory leakage in modern hardware, surfaced from an analysis of methods that increase processing speeds.

Spectre can be defined as a “side channel attack”. As the name suggests, side channel attacks exploit *observable side effects*, e.g., time, power consumption, of computational processes to gain information about critical data that is “otherwise unavailable” (Kocher 1). In the case of Spectre, a timing attack can be performed on maliciously obtained data; that is, an attacker can measure and analyze the time to complete a specific instruction compared to an understood, attacker-defined control instruction to detect anomalies and to make conclusions about data. Specifically, the variant of Spectre discussed in this paper utilizes a timing attack method that can probe and decode victim memory through a CPU cache side channel (Kocher 1).

CPU caching is the storage of data retrieved from memory that the processor deems likely to be reused. Retrieving data from memory (Dynamic RAM) is significantly slower than retrieval from cache memory (Static RAM); therefore, caching increases retrieval time. x86 systems typically utilize three different cache locations, named L1, L2, and L3 cache; ordered from fastest to slowest, respectively (“Optimization Manual” 2-11 – 2-51). Before retrieving data from DRAM, the CPU will first check to see if L1 cache contains the data required. If not stored in L1 cache, the processor will check L2 cache, then L3. If data is not found in any of the cache locations, a *cache miss* is incurred, and DRAM will be accessed (Kocher 4). Cache misses cause a measurable delay that an attacker can use to determine if data is stored in cache or not. Cache can be indirectly manipulated; a “Flush+Reload” attack involves manual cache flushing, followed by deliberate reloading of cache to infer a normally restricted value (Kocher 2). The *clflush* x86 command performs a cache flush for x86 machines ("Developer's Manual" Vol. 1, 5-8). Data can be cached for a variety of purposes, such as for *speculatively executed* instructions.

Speculative execution is a functionality of modern CPUs used to avoid time delays in pipelined systems. When a conditional branching statement is encountered by the CPU, a read from DRAM is often required. As mentioned previously, a memory read is significantly slower than L1 cache retrieval; as such, a pipelined system will bottleneck if it must wait for DRAM retrieval. To avoid bottlenecks, *branch prediction* is performed to "guess" which branch will be executed before the branch's conditional statement is evaluated. This process includes proprietary implementations unavailable to the public, however, some of the functionality has been discovered. For example, AMD uses an artificial intelligence neural network for branch prediction, and Intel has been found to use Branch Target Buffers (BTB) that map starting point and destination addresses for branching statements previously executed (Kocher 12; Bhattacharya 6). If a branch is predicted as likely to be run by branch predictors, the statements inside are speculatively executed *before* the Boolean expression is resolved. This requires the state of the program prior to the speculative execution to be saved in registers in case the prediction was incorrect. Many Intel machines store speculatively executed statement results in L1 cache ("Optimization Manual" 2-25). Kocher's research has proved that this includes results of illegal operations, such as accessing outside the bounds of an array (Kocher 5). Timing attacks can be used to decode cache storage results from speculatively executed code ("Intel Analysis" 2-3). It is important to note branch predictions are trainable and can be manipulated by an attacker to perform incorrect predictions for specific branches, which is further detailed below.

To exploit Spectre, an attacker may start by flushing the cache, then initializing a first array that contains arbitrary contents (call it *A*), and a second array that contains character mappings (call it *instrument*). Branch predictors in the architecture are then deliberately trained to predict a specific if statement to be true (Kocher 2). Next, an attacker may deliberately run a malicious branch in place of the previously correct branch to trick the branch predictors into improperly performing speculative execution. The speculatively executed statements in the malicious branch can attempt to store a reference to *instrument[A[malicious index]]* (which points to victim memory), generating an out of bounds exception (for *A*) *after* the element of *instrument* (assuming a valid index of *instrument*) is accessed and cached by the processor, using *A[malicious index]* returned as an index for *instrument*. If *A[malicious index]* returns a valid index for *instrument*, the victim memory contents have been leaked into the malicious user-space via a cache side channel (Kocher 5). Then an attacker can iterate through *instrument*, measure retrieval time, and look for an outlier of extremely fast retrieval time (a cache hit, i.e., opposite of cache miss) that implies the element is stored in CPU cache, and that the victim data is, in fact, the index of the element in *instrument* triggering the cache hit (Kocher 6). This process can be repeated and extrapolated to retrieve more complete parts of victim memory. The described procedure is only a single variant of Spectre to illustrate the vulnerability; further exploitation detail has been credibly established by Google Project Zero, such that one may reliably determine individual bits in memory, rather than a full byte at a time for reliable results (Horn).

Spectre has been verified to work on an enormous amount of architectures, “including Ivy Bridge, Haswell and Skylake based processors […] AMD Ryzen CPUs […] [and] Samsung and Qualcomm processors (which use an ARM architecture)” (Kocher 12). Mitigation of the vulnerability will be a long-term, multi-faceted effort by engineers across many domains.

Works Cited

Kocher, Paul, et al. "Spectre Attacks: Exploiting Speculative Execution." arXiv preprint arXiv:1801.01203 (2018).

Horn, Jann. "Reading Privileged Memory With A Side-Channel". Googleprojectzero.Blogspot.Com, 2018, https://googleprojectzero.blogspot.com/2018/01/reading-privileged-memory-with-side.html. Accessed 7 May 2018.

Intel Corporation. "Intel Analysis of Speculative Execution Side Channels." Intel white paper; Document Number: 336983-001 (2018).

Intel Corporation. "Intel® 64 and IA-32 Architectures Software Developer's Manual." Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D and 4 (2018).

Intel Corporation. "Intel 64 and IA-32 Architectures Optimization Reference Manual." Order Number: 248966-033 (2016).

Bhattacharya, S., et al. "Template attack on blinded scalar multiplication with asynchronous perf-ioctl calls." Cryptology ePrint Archive, Report 2017/968, (2017).