White Paper | AMD64 TECHNOLOGY

SPECULATIVE STORE BYPASS DISABLE

5.21.18

SPECULATIVE STORE BYPASS DISABLE

As specified in the AMD64 Architectural Programmer's Manual Vol.2 chapter 7 on Memory System, reads from memory marked with the proper memory type can read memory out-of-order, speculatively and before writes have occurred. This means there are occurrences where a load can read and pass forward in a speculative manner previous values of the memory location. The processor has logic to correct this occurrence and provide the proper in-order response to the program. However, speculation may occur based on that data and cause different cache lines to be accessed. The previous value used by the processor will be a value that is within the context and privilege level of the current program. This document describes a feature designed to mitigate speculation of load instruction past older store instructions with an unknown address.

AMD recommends leaving memory disambiguation on as the default setting.

PRESENCE

This feature is implemented in 2 different ways on AMD processors. For future processors, a speculative store bypass disable (SSBD) bit is present in bit 2 of SPEC_CTL (MSR 48h). This is indicated by presence of CPUID Function 8000_0008, EBX[24]=1. When SPEC_CTL bit 2 is set, the processor keeps loads from speculating around older stores. For some legacy usage, there is also a bit 2 in a new VIRT_SPEC_CTRL MSR (MSR C001_011F). When VIRT_SPEC_CTL bit 2 is set, the processor keeps loads from speculating around older stores. It is indicated by CPUID function 8000_0008, EBX[25]=1. AMD also provides a CPUID bit to indicate when speculative store bypass disable is no longer needed to prevent speculative loads bypassing older stores on future processors. This is indicated by CPUID Function 8000_0008, EBX[26]=1.

USAGE

On AMD processors that support both SPEC_CTL and VIRT_SPEC_CTL, if bit 2 (SSBD) is 1 in either MSR the processor keeps loads from speculating around older stores. However, it is preferred that in this scenario the software use the SPEC_CTL version only. Software should follow the following enumeration path to determine how to control SSBD on a given processor.

```
If (Fn8000_0008 EBX[26] =1)  ; SSBD explicitly stated as not needed
   Return;
Else if (Fn8000_0008 EBX[24] =1 ) ; SSBD supported in SPEC_CTL bit 2 MSR 48
    {Set MSR 0x48 bit 2 with desired value;
     Return;
}
Else if (Fn8000_0008 EBX[25] =1 ) ; Use VIRT_SPEC_CTL bit 2 MSR C001_011F
   {Set MSR c001_011F bit 2 with desired value   ;
     Return;
   }
Else
     { Use non-architectural means (described later)
       Return;
           }
```

Both SPEC_CTRL and VIRT_SPEC_CTRL are not architecturally serializing WRMSRs. They are still execution serializing and prevent any execution of future instructions until they have completed.

For processors that support SPEC_CTRL bit 2 (SSBD) but don't support either bit 1 or bit 0, the processor will still allow reads and writes to bits 1 and 0 even though they don't change processor behavior. Any attempt to write bits 63:3 to a non-zero value will cause a Gp fault.

For processors that support VIRT_SPEC_CTRL bit 2, writes are ignored to the non-supported bits and reads are read as zero.

HYPERVISOR USAGE MODELS

In a similar method to the IBC use of SPEC_CTL bits 0 and 1, there is a host and guest version of SPEC_CTL for bit 2. When the guest is running they are logically or-ed together to control the policy in effect for the guest. When running in host mode only the host value applies. This keeps the needs of the guest from affecting the host as well as giving the host absolute control for disablement without intercepts.

Because some AMD processors only support a non-architectural means to allow SSBD to be enabled, hypervisor's can virtualize the existence of VIRT_SPEC_CTRL to help deal with the lack of an available architectural mechanism. This is accomplished by presenting the guest OS with the CPUID bit Fn8000_0008 EBX[25] =1. Then when the guest requires SSBD support it will write VIRT_SPEC_CTRL which can be intercepted to the hypervisor and the hypervisor can write the non-architectural bit or update the host or guest version of SPEC_CTL bit 2 if the processor supports it. This allows the traditional SPEC_CTL to remain as an MSR that does not need to be intercepted by the hypervisor. The hypervisor should continue to intercept the non-architectural MSR (see Non-architectural MSR section) in the guest and inject a GP fault if the guest tries to manipulate it directly.

NON-ARCHITECTURAL MSRS

AMD processors families 15h, 16h, and some models of family 17h have logic that allow loads to bypass older stores but lack the architectural SPEC_CTRL or VIRT_SPEC_CTRL. For software that requires protection, software can directly manipulate non-architectural MSRs.

For Family 15h processors, to enable SSBD , software should set bit 54 of MSR C001_1020 and to disable SSBD, software should clear the same bit. The MSR is per logical processor and can be saved and restored for software context switch. All other bits should be left in the state observed on a read.

For Family 16h processors, to enable SSBD, software should set bit 33 of MSR C001_1020 and to disable SSBD, software should clear the same bit. The MSR is per logical processor and can be saved and restored for context switch. All other bits should be left in the state observed on a read.

For Family 17h processors, to enable SSBD, software should set bit 10 of MSR C001_1020. If the family 17h processor only supports 1 logical processor (CPUID fn8000_001E.EBX[15:8] =0), software should clear bit 10 of MSR C001_1020 to disable SSBD. The MSR is per logical processor and can be saved and restored for context switch. All other bits should be left in the state observed on a read. However, if the family 17h processor supports 2 logical processors (CPUID fn8000_001E.EBX[15:8] =1), this MSR is shared between the 2 logical processors on each logical core. Therefore, software should manage the SSBD needs of both logical processors to prevent one logical processor from disabling SSBD when it is required for the other logical processor. Here is an example for management, other optimization techniques can be applied:

```
If (Family 17h and CPUID fn8000_001E.EBX[15:8] = 1)   ; Family 17 with 2 logical processors per
  {
    If (SSBD[my_logical_processor] != newval)    ; value changing,
    {
     acquire_lock;
     If (newval==1)                  ; Enabling SSBD
           {
            SSBD[my_logical_processor]=1;   ; set state of SSBD for this logical processor
            If (SSBD[my_sibling_processor]==0)    ; bit 10 may not be set already
              {
                Rdmsr(val, C001_1020);
                 If (val[10]=0)                  ; check bit 10 before writing as write is expensive.
                   {
                    val[10]=1;
                    Wrmsr(val, C001_1020);
                   }
               }
           }
    Else  ;                          disabling SSBD
         {
           SSBD[my_logical_processor]=0;        ; set my desired state of SSBD
            If (SSBD[my_sibling_processor]==0)    ; bit 10 can be cleared if both threads agree.
              {
                Rdmsr(val, C001_1020)
                 If (val[10]=1)
                    {
                     Val[10]=0;
                    Wrmsr(val, C001_1020);
                      }
               }
           }
  Release_lock;
   }
 ;  values were the same, nothing to do
}
```

5.21.18

www.AMD.com/securityupdates

AMD