AMD

White Paper | SOFTWARE TECHNIQUES FOR MANAGING SPECULATION ON AMD PROCESSORS

## INTRODUCTION

*Speculative execution is a basic principle of all modern processor designs and is critical to support high performance hardware. Recently, researchers have discussed techniques to exploit the speculative behavior of x86 processors and other processors to leak information to unauthorized code*. This paper describes software options to manage speculative execution on AMD processors** to mitigate the risk of information leakage. Some of these options require a microcode patch that exposes new features to software.*

*The software exploits have recently developed a language around them to make them easier to reference so it is good to review them before we start discussing the architecture and mitigation techniques.*

## VARIANT DESCRIPTIONS

A software technique that can be exploited is around software checking for memory references that are beyond the software enforced privileged limit of access for the program (bounds checking). In a case where the maximum allowed address offset for a data structure is in memory, it can take a large number of processor cycles for the processor to obtain the maximum allowed address. This opens up the window of time where speculative execution can occur while the processor is determining if the address is within the allowed range. If the out of range address is not constrained in the speculative code path based on the way the code is written, the processor may speculate and bring in cache lines that are currently allowed to be referenced based on the privilege of the current mode but outside the boundary check. This is referred to as variant 1 (Google Project Zero and Spectre) and an example of the code can be observed in mitigation V1-1. For variant 1 mitigation, AMD is recommending software only solutions which need to be evaluated in a wide range of software including kernel software, JITs, browsers, and other user applications.

Another technique that can be exploited by software is indirect branches. Indirect branches are supported in x86 with the ability for software to branch to instruction targets that are loaded in a register, a value loaded directly from memory, or a return instruction from a previous subroutine call. The branch prediction structures vary per processor implementation and therefore the techniques allowing lesser privileged code to interfere with the indirect branch predictor also vary. In an architecture where the processor can predict an incorrect target and it can take a large number of cycles to determine the correct target, this opens up a window for a speculative execution attack. This is referred to as variant 2 (Google Project Zero and Spectre) and an example can be seen in mitigation V2-1. For variant 2, there are both software and software plus hardware mitigations.

A third technique is based on a software performance optimization. Software running in a lesser privilege mode typically has page table mappings for more privileged code present in the page table context that is running. This allows for high performance switching between the two modes and the software uses extra page table attributes enforced by the hardware to restrict access to the privileged data when in lesser privileged modes. However, on some processors it has been observed that if software accesses the more privileged data when the processor is in a lesser privileged mode, the architectural fault may be delayed. This opens up a window for a speculative execution attack where privileged data is then forwarded to subsequent instructions for speculative execution. This is referred to as a variant 3 (Google Project Zero and Meltdown). No AMD processor has been designed with this behavior and so we are not discussing mitigation steps in the rest of the document for this variant but we are including it here for completeness. Software developers should use CPUID vendor ID checks to identify AMD processors to avoid implementing variant 3 mitigations.

** See http://www.amd.com/en/corporate/speculative-execution for more information.*

*** In this document the term processor refers to x86 code executing on AMD CPUs and APUs.*

To mitigate the above described variants, there are a variety of possible techniques software can use. Because unique tools may be preferred for different applications, this document discusses a number of potential mitigations on AMD processors. Due to the variety of software architectures and requirements, there is no single one-size-fits-all solution to mitigating this type of information leakage. Throughout this document, potential mitigations are noted as follows with a V1 or V2 prefix to indicate which variant they are targeting or a G prefix which means they are applicable for both:

***MITIGATION*** <#>
**Description:** <Description of mitigation>
**Effect:** <Effect on CPU hardware>
**Applicability:** <Notes on specific AMD processors which can/cannot use this mitigation>

Each mitigation technique will have different performance characteristics (including potential negative impacts to the performance of the system), and software developers must evaluate which mitigation solution(s) are the best fits for their specific needs. Please also note that while some mitigations presented here may work on non-AMD processor architectures, AMD has only evaluated their behavior on AMD processors.

## MITIGATIONS

***MITIGATION G-1***
**Description:** Clear out untrusted data from registers (e.g. write 0) when entering more privileged modes or sensitive code.
**Effect:** By removing untrusted data from registers, the CPU will not be able to speculatively execute operations using the values in those registers.
**Applicability:** All AMD processors

Instructions that cause the machine to temporarily stop inserting new instructions into the machine for execution and wait for execution of older instructions to finish are referred to as dispatch serializing instructions.

***MITIGATION G-2***
**Description:** Set an MSR in the processor so that LFENCE is a dispatch serializing instruction and then use LFENCE in code streams to serialize dispatch (LFENCE is faster than RDTSCP which is also dispatch serializing). This mode of LFENCE may be enabled by setting MSR C001_1029[1]=1.
**Effect:** Upon encountering an LFENCE when the MSR bit is set, dispatch will stop until the LFENCE instruction becomes the oldest instruction in the machine.
**Applicability:** All AMD family 10h/12h/14h/15h/16h/17h processors support this MSR. LFENCE support is indicated by CPUID function1 EDX bit 26, SSE2. AMD family 0Fh/11h processors support LFENCE as serializing always but do not support this MSR. AMD plans support for this MSR and access to this bit for all future processors.

***MITIGATION V1-1***

**Description:** With LFENCE serializing, use it to control speculation for bounds checking. For instance, consider the following code:

```
1:   cmp   eax, [buffer_top]      ; compare eax (index) to upper bound
2:   ja    out_of_bounds          ; if greater, index is too big
3:   mov   ebx, [eax]             ; read buffer
```

In this code, the CPU can speculative execute instruction 3 (mov) if it mispredicts the branch at 2 (ja). If this is undesirable, software should implement:

```
1:   cmp   eax, [buffer_top]      ; compare eax (index) to upper bound
2:   ja    out_of_bounds          ; if greater, index is too big
3:   lfence                       : serializes dispatch until branch
4:   mov   ebx, [eax]             ; read buffer
```

**Effect:** In the second code sequence, the processor cannot execute op 4 because dispatch is stalled until the branch target is known.
**Applicability:** Applies to all AMD processors

***MITIGATION V1-2***

**Description:** Create a data dependency on the outcome of a compare to avoid speculatively executing instructions in the false path of the branch. For instance, consider the following code:

```
1:   cmp   eax, [buffer_top]      ; compare eax (index) to upper bound
2:   ja    out_of_bounds          ; if greater, index is too big
3:   mov   ebx, [eax]             ; read buffer
```

In this code, the CPU can speculative execute instruction 3 (mov) if it mispredicts the branch at 2 (ja). If this is undesirable, software should do:

```
1:   xor   edx, edx
2:   cmp   eax, [buffer_top]      ; compare eax (index) to upper bound
3:   ja    out_of_bounds          ; if greater, index is too big
4:   cmova eax, edx               ; NEW: dummy conditional mov
5:   mov   ebx, [eax]             ; read buffer
```

**Effect:** In the second code sequence, the processor cannot execute op 4 (cmova) because the flags are not available until after instruction 2 (cmp) finishes executing. Because op 4 cannot execute, op 5 (mov) cannot execute since no address is available.
**Applicability:** Applies to all AMD processors

***MITIGATION V1-3***

**Description:** Create a data dependency on the outcome of a compare to mask the array index to keep it within bounds. For instance, consider the following code:

```
1:    cmp   eax, [buffer_top]      ; compare eax (index) to upper bound
2:    ja    out_of_bounds          ; if greater, index is too big
3:    mov   ebx, [eax]             ; read buffer
```

In this code, the CPU can speculatively execute instruction 3 (mov) if it mispredicts the branch at 2 (ja). If this is undesirable, software should do:

```
1:    cmp   eax, [buffer_top]      ; compare eax (index) to upper bound
2:    ja    out_of_bounds          ; if greater, index is too big
3:    and   eax, $MASK             ; NEW: Mask array index
4:    mov   ebx, [eax]             ; read buffer
```

**Effect:** In the second code sequence, the processor will mask the array index before the memory load constraining the range of addresses that can be speculatively loaded. For performance it is best if $MASK is an immediate value.

**Applicability:** Applies to all AMD processors. This mitigation works best for arrays that are power-of-2 sizes but can be used in all cases to limit the range of addresses that can be loaded.

In the case of RET instructions, RIP values are predicted using a special hardware structure that tracks CALL and RET instructions called the return stack buffer. Other indirect branches (JMP, CALL) are predicted using a branch target buffer (BTB) structure. While the mechanism and structure of this buffer varies significantly across AMD processors, branch predictions in these structures can be controlled with software changes to mitigate variant 2 attacks.

***MITIGATION V2-1***

**Description:** Convert indirect branches into a "retpoline". Retpoline sequences are a software construct which allows indirect branches to be isolated from speculative execution. It uses properties of the return stack buffer (RSB) to control speculation. The RSB can be filled with safe targets on entry to a privileged mode and is per thread for SMT processors. So instead of

```
1:    jmp *[eax]                   ; jump to address pointed to by EAX2:
```

To this:

```
1:    call l5 ; keep return stack balanced
l2:   pause ; keep speculation to a minimum
3:    lfence
4:    jmp l2
l5:   add rsp, 8 ; assumes 64 bit stack
6:    push [eax] ; put true target on stack
7:    ret
```

and this 1: call *[eax] ;

***MITIGATION V2-1** (CONTINUED)*

To this:

```
 1:    jmp l9
l2:    call l6                  ; keep return stack balanced
l3:    pause
 4:    lfence                    ; keep speculation to a minimum
 5:    jmp l3
l6:    add  rsp, 8              ; assumes 64 bit stack
 7:    push [eax]               ; put true target on stack
 8:    ret
L9:    call l2
```

**Effect:** This sequence controls the processor's speculation to a safe known point. The performance impact is likely greater than V2-2 but more portable across the x86 architecture. Care needs to be taken for use outside of privileged mode where the RSB was not cleared on entry or the sequence can be interrupted. AMD processors do not put RET based predictions in BTB type structures.

***MITIGATION V2-2***

**Description:** Convert an indirect branch into a dispatch serializing instruction sequence where the load has finished before the branch is dispatched. For instance, change this code:

```
1:    jmp *[eax]               ; jump to address pointed to by EAX
```

To this:

```
1:    mov eax, [eax]           ; load target address
2:    lfence                   ; dispatch serializing instruction
3:    jmp *eax
```

**Effect:** The processor will stop dispatching instructions until all older instructions have returned their results and are capable of being retired by the processor. At this point the branch target will be in the general purpose register (eax in this example) and available at dispatch for execution such that the speculative execution window is not large enough to be exploited.

**Applicability:** All AMD processors. AMD plans that this sequence will continue to work on future processors until support for other architectural means to control indirect branches are introduced.

***MITIGATION V2-3***
**Description:** Execute a series of CALL instructions upon entering more privileged code to fill up the return address predictor.
**Effect:** The processor will only predict RET targets to the RIP values in the return address predictor, thus preventing attacker controlled RIP values from being predicted.
**Applicability:** All AMD processors. The size of the return address predictor varies by processor, all current AMD processors have a return address predictor with 32 entries or less. Future processors that have more than 32 RSB entries are planned to be architected to not require software intervention.

***MITIGATION V2-4***
**Description:** An architectural mechanism, Indirect Branch Control (IBC), is being added to the x86 ISA to help software control branch prediction of jmp near indirect and call near indirect instructions. It consists of 3 features: Indirect Branch Prediction Barrier (IBPB), Indirect Branch Restricted Speculation (IBRS) and Single Thread Indirect Branch Predictors (STIBP).
**Effect:** These features give software another mechanism through architectural MSRs to provide mitigation for different variant 2 exploits.

- **IBPB –** Places a barrier such that indirect branch predictions from earlier execution cannot influence execution after the barrier.
- **IBRS –** Restricts indirect branch speculation when set.
- **STIBP –** Provides sibling thread protection on processors that require sibling indirect branch prediction protection

**Applicability:** As a new feature, these mechanism are available in only a limited number of current AMD processors and require a microcode patch. These 3 features are individually enumerated through CPUID and all processors do not support all features. These features also require software updates to write the MSR where appropriate.

After a RIP value is predicted, the new RIP value is sent through a TLB and table walker pipeline before instruction bytes can be fetched and sent for execution.

***MITIGATION G-3***
**Description:** Enable Supervisor Mode Execution Protection (SMEP).
**Effect:** The processor will never speculatively fetch instruction bytes in supervisor mode if the RIP address points to a user page. This prevents the attacker from redirecting the kernel indirect branch to a target in user code.
**Applicability:** All AMD processors that support SMEP (Family 17h, Family 15h model >60h)

The load-store unit is a key area for controlling speculation because information leakage comes from the residual nature of cache lines after a speculative fill.

***MITIGATION G-4***
**Description:** Enable SMAP (Supervisor Mode Access Protection)
**Effect:** The processor will never initiate a fill if the translation has a SMAP violation (kernel accessing user memory). This can prevent the kernel from bringing in user data cache lines. With SMEP and SMAP enabled the attacker must find an indirect branch to attack in the area marked by SMAP that is allowed to access user marked memory.
**Applicability:** All AMD processors which support SMAP ( family 17h and greater).

## CONCLUSION

There are a variety of techniques software can use for managing processor speculation, each with different properties and trade-offs. Some techniques involve managing what addresses the processor can use for speculative instruction fetch, stopping the dispatch or execution of speculative instructions, or managing what data addresses the processor can calculate. In addition, newer and future AMD products support additional security features (such as SMEP, SMAP, IBC) which are particularly useful in controlling speculation across kernel/user privilege boundaries.

AMD is aligned with the x86 community that V1-1 (lfence) is the preferred variant 1 software solution and that the V2-1 (retpoline) is the preferred variant 2 software solution. AMD continues to evaluate opportunities for new mitigations in both the x86 ISA and micro-architecture for future AMD processors.

## REFERENCES:

*APM Volume 2: http://support.amd.com/TechDocs/24593.pdf*
*Processor Programming Reference (PPR) for family 17h: http://support.amd.com/TechDocs/54945_PPR_Family_17h_Models_00h-0Fh.pdf*

**AMD.com/speculative_execution**