Hardware Features and Behavior Related to Speculative Execution

Technical Paper

Revision 2.0

January 2026

Document Number: 758913

Public URL: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/hardware-behavior-related-to-speculative-execution.html

Revision History

Date	Revision	Description
April 2024	1.0	Initial public version.
September 2024	1.1	Added section 3.3 for miscellaneous speculation and section 7.6 for Linear Address Masking and Linear Address Space Separation (LAM/LASS).
Feb 2025	1.2	Added information on SMEP/SMAP
September 2025	1.3	Added information for Intel Advanced Performance Extensions (Intel© APX) in section 7.8, updated bounds check bypass guidance and revised bounds clipping in section 3.2.4, and moved OS and VMM indirect branch guidance to section 3.
December 2025	1.4	Clarified additional software guidance.
January 2026	2.0	Second public version.

§

Contents

1.0 Introduction 6

2.0 Speculative Execution 7

2.1 Incidental Channels 8

2.2 Restricting Speculative Execution 8

3.0 Control-Flow Speculation 10

3.1 Indirect Branches 10

3.1.1 Overview of Indirect Branch Predictors 10

3.1.2 Indirect Branch Speculation Control Mechanisms 11

3.1.3 Software Techniques for Indirect Speculation Control 18

3.1.4 Guidance for Managing Indirect Branches 19

3.2 Conditional Branches 20

3.2.1 Overview of Bounds Check Bypass 20

3.2.2 Identifying Bounds Check Bypass Vulnerabilities 22

3.2.3 Conditional Branch Speculation Analysis 27

3.2.4 Software Techniques for Conditional Speculation Control 28

3.2.5 Operating System Mitigations 34

3.3 Speculation within Instructions 36

4.0 Data Speculation 38

4.1 Overview of Data Speculation 38

4.2 Speculative Store Bypass 38

4.3 Speculative Store Bypass Control Mechanisms 39

4.3.1 Software-Based Mitigations 39

4.3.2 Speculative Store Bypass Disable (SSBD) 40

4.4 Fast Store Forwarding Predictor 41

4.4.1 Potential for Transient Execution Disclosure Gadgets 41

4.5 Fast Store Forwarding Predictor Controls 41

4.5.1 Cross-Domain and Cross-Thread Training Isolation 41

4.5.2 Predictive Forwarding Barriers 42

4.5.3 FSFP Disabled Modes 42

4.5.4 Software Guidance 42

5.0 Data-Dependent Prefetchers 44

6.0 Additional Software Guidance 45

7.0 Related Intel Security Features and Technologies 46

7.1 Intel® OS Guard 46

7.2 Execute Disable Bit 46

7.3 Intel® Control flow Enforcement Technology (Intel® CET) 46

7.3.1 CET Shadow Stack Speculation Limitations 46

7.3.2 Intel CET Indirect Branch Tracking (CET IBT) Speculation Limitations 47

7.4 Protection Keys 47

7.5 Supervisor-Mode Execution Prevention (SMEP) 47

7.6 Supervisor-Mode Access Prevention (SMAP) 47

7.7 Linear Address Space Separation 48
7.8 Intel® Advanced Performance Extensions (Intel® APX) 48

8.0 CPUID Enumeration and Architectural MSRs 50

9.0 Resources 51

10.0 Appendix: Summary Table of How Mitigation Techniques are Used 52

1.0 Introduction

Modern processors use speculative execution to provide higher performance, more efficient resource utilization, and better user experiences. The speculation mechanisms may use various forms of predictors to anticipate future program execution and improve performance by having instructions execute earlier than their program order. While these predictors are designed to have high accuracy, wrong predictions can occur and result in mis-speculation, where a processor first executes instructions based on a prediction, and later squashes them to return to correct program execution. An attacker can potentially exploit such mis-speculation to reveal sensitive data in a transient execution attack.

While previous documentation described specific speculative execution vulnerabilities and their mitigations, this article consolidates the prior guidance on speculative execution with better organization, along with clarifications and additional guidance to help readers navigate this topic. We continue to refer to the per-vulnerability guidance documents for more details.

This consolidated document explains how to effectively manage some common forms of speculation in Intel processors, limit the performance impact of mitigations, and avoid mitigation redundancies for the features and behaviors included. It also provides an overview of some of the different types of speculation on current Intel processors and describes the hardware controls and software-based techniques that developers can use to restrict speculation and, where relevant, reduce the ability of potential adversaries to infer secret data due to speculation. Intel plans to update this document periodically to incorporate new guidance documents as they are released; for example, reflecting speculation mechanisms that may be added on future Intel processors.

This document is organized as follows: Section 2.0 Speculative Execution starts with an introduction to speculative execution, describes control-flow and data speculation, and outlines options to restrict speculation. Section 3.0 Control-Flow Speculation details control-flow speculation due to indirect and conditional branches and techniques to restrict control-flow speculation on Intel processors. Section 4.0 Data Speculation describes variants of data speculation such as memory disambiguation and options to manage data speculation. Section 5.0 Data-Dependent Prefetchers outlines data-dependent prefetches. Section 6.0 Additional Software Guidance summarizes the recommendations for restricting speculation in common use cases, such as, after a processor enters a higher privilege level. Section 7.0 Related Intel Security Features and Technologies describes security features and technologies which reduce the effectiveness of malicious attacks described in the previous sections. Section 8.0 CPUID Enumeration and Architectural MSRs references the processor enumerations and model-specific registers that provide the hardware features and mechanisms described in this document.

2.0 Speculative Execution

In order to improve performance, modern processors make predictions about the program's future execution. Processors use these predictions to speculatively execute younger instructions ahead of the current instruction pointer. As the processor advances in program execution, it resolves all conditions required to determine the correctness of the prediction. If the original predictions were correct, the speculatively executed instructions can retire, and their state becomes architecturally visible. If a prediction was wrong, the instructions which were speculatively executed based on the misprediction must be squashed and do not affect architectural states. These squashed instructions, which were only executed speculatively, are called *transient instructions*. Based on the resolved conditions, the processor then resumes with the correct program execution. A more detailed description of speculative execution is available in the Refined Speculation Execution Terminology article.

Processors implement various forms of predictions and speculation which may result in instructions being speculatively executed, including:

- Control-flow speculation involves speculatively executing instructions based on a prediction of the program's control flow.
    - Indirect branch predictors predict the target address of indirect branch instructions[1] to allow instructions at the predicted target address to be speculatively executed before the target address has been resolved.
    - Conditional branch predictors predict the direction of conditional branches to allow instructions on the predicted path to be speculatively executed before the condition has been resolved.
- Data speculation involves speculatively executing instructions which depend on the values from previous instructions before the previous instructions have been executed. For example, the processor may speculatively forward data from a previous load to younger dependent instructions before the addresses of all intervening stores are known.

Speculation can also occur for other reasons, and in particular, processors may speculate that architectural or microarchitectural events (for example, exceptions or assists) do not occur. This may result in instructions being transiently executed and squashed later by the processor when an event is handled[2].

While speculative execution predictors strive to have high accuracy, predictions can be wrong. A malicious actor may be able to use mispredictions to perform transient execution attacks, in which case a malicious actor may attempt to retrieve secret information from transiently executed instructions through an incidental channel.

[1] The specific instructions are described in section 3.1.1 Overview of Indirect Branch Predictors. Note that the target address of direct branch instructions is also predicted but Intel processors do not allow speculative execution at incorrect target addresses that are due to direct branches.
[2] Vulnerabilities such as Rogue Data Cache Load, Rogue System Register Read, L1 Terminal Fault, and Lazy FP may allow malicious actors to leverage such speculative execution to bypass existing security restrictions and infer secret data on some processors. Refer to the respective technical papers and section 9.0 Resources for more details.

Multiple sources of speculation may affect the same instruction. For example, an indirect branch may be affected by both control-flow speculation and data speculation. Control-flow speculation may cause the indirect branch to be predicted with a target based on past behavior. Data speculation could later affect the indirect branch's source data and cause it to transiently go to an incorrectly predicted location before later redirecting to the correct location. We use the term *attacker-controlled jump redirection* where a malicious actor controls the speculative branch target (through data speculation), to distinguish this from *attacker-controlled prediction* where a malicious actor controls the *predicted* branch target (through control-flow speculation).

2.1 Incidental Channels

There are several sources of incidental channels that may be used to retrieve information from transiently executed instructions. An overview of possible incidental channels is provided in the incidental channel taxonomy.

Using such incidental channels, a malicious actor may be able to gain information through observing certain states of the system, such as by measuring the microarchitectural properties of the system. Unlike buffer overflows and other vulnerability classes, incidental channels do not directly influence the execution of the program, nor allow data to be modified or deleted.

For instance, a cache timing side channel involves an adversary detecting whether a piece of data is present in any or a specific level of the processor's caches, which may be used to infer some other related information. One common method to detect whether the data of interest is present in a cache is to use timers to measure the latency to access memory at the corresponding address and compare with the baseline timing of memory accesses that hit the cache or memory.

2.2 Restricting Speculative Execution

System operators have a range of options available to restrict speculation in Intel processors and reduce the risk of transient execution attacks. Intel processors provide several controls, such as enhanced Indirect Branch Restricted Speculation (IBRS) and Speculative Store Bypass Disable (SSBD), to restrict control speculation of indirect branches and to control data speculation, respectively. Section 3.1.2 Indirect Branch Speculation Control Mechanisms details the indirect branch speculation controls available and their usage and Section 4.3 Speculative Store Bypass Control Mechanisms describes controls to restrict data speculation.

Speculation can also be restricted through software-based techniques: For example, software can use a technique called retpoline (see Section 3.1.3 Software Techniques for Indirect Speculation Control) to restrict indirect branch speculation and use bounds clipping to prevent speculative out-of-bounds array accesses following conditional branches (refer to Section 3.2.1 Overview of Bounds Check Bypass).

More generally, software can insert speculation-stopping barriers at the proper locations as needed to prevent a speculative side channel. The `LFENCE` instruction, or any serializing instruction, can serve as such a barrier. The `LFENCE` instruction and serializing instructions ensure that no later instruction will execute, even speculatively, until all prior instructions have completed locally. The `LFENCE` instruction has lower latency than the serializing instructions and thus is recommended when a speculation-stopping barrier is needed.

Certain security features with architectural effect can also be effective with respect to speculative execution. For example, when Supervisor Mode Access Prevention (SMAP) is enabled, supervisor loads executed with a cleared AC flag will not transiently access memory in user mode pages from CPL0. This may prevent an attacker from using user memory for an incidental channel.

3.0 Control-Flow Speculation

As highlighted in Section 2.0 Speculative Execution, control-flow speculation occurs when the processor speculatively executes instructions based on control flow prediction. The two main sources of transient execution related to control-flow speculation on Intel processors are indirect branches and conditional branches. In some conditions, certain non-branch instructions may also have speculation in the internal flow of their implementation.

The following section of this document describes control-flow speculation due to indirect branches, conditional branches, and other miscellaneous situations, as well as the hardware and software mechanisms that can be used to restrict such speculation.

3.1 Indirect Branches

3.1.1 Overview of Indirect Branch Predictors

Intel processors use *indirect branch predictors* to determine the target address of instructions that are to be speculatively executed after a near indirect branch instruction, as enumerated in the table below.

Table 1. Instructions that use Indirect Branch Predictors

Branch Type	Instruction	Opcode
Near Call Indirect	`CALL r/m16, CALL r/m32, CALL r/m64`	`FF /2`
Near Jump Indirect	`JMP r/m16, JMP r/m32, JMP r/m64`	`FF /4`
Near Return	`RET, RET Imm16`	`C3, C2 Iw`

References in this document to indirect branches are only to near call indirect, near jump indirect and near return instructions.

To make accurate predictions, indirect branch predictors are trained through program execution. Specifically, indirect branch predictors learn the target addresses of indirect branch instructions when they execute and use them for target prediction of subsequent execution of indirect branch instructions. While being accurate for most cases, misprediction may happen and the indirect branch predictor may predict the wrong target address, which can result in instructions at an incorrect code location being speculatively executed and later squashed.

Intel processors implement different forms of indirect branch predictors, which could include (but are not necessarily limited to):

- IP-based predictors that predict the indirect branch target address based on the address of the branch instruction.

- History-based predictors that predict the indirect branch target address based on the history of previously executed branch instructions. This allows the processor to predict different targets for the same indirect branch depending upon the previous code leading up to the indirect branch. For example, this could be based on a Branch History Buffer (BHB) which holds history used to select branch targets in these predictors.

On current Intel processors, an important specific example of a predictor is the Return Stack Buffer (RSB) that predicts the targets of near `RET` instructions based on previous corresponding `CALL` instructions. Each execution of a near `CALL` instruction with a non-zero displacement adds an entry to the RSB that contains the address sequentially following that `CALL` instruction. The RSB is not used or updated by far `CALL`, far `RET`, or `IRET` instructions.

Note that besides control-flow speculation, such as in indirect branch predictions, data speculation can also be the origin of speculative execution in the context of indirect branch instructions. For instance, due to memory disambiguation, an indirect jump instruction may load the target address from a memory location and speculatively jump to this target address before an older store instruction has stored a different target address to that memory location[3].

Branch Target Injection (BTI), Branch History Injection (BHI), and Intra-mode BTI are all microarchitectural transient execution attack techniques which involve an adversary influencing the target of an indirect branch by training the indirect branch predictors. Intel processors support indirect branch speculation control mechanisms which can be used to mitigate such attacks.

3.1.1.1 Indirect Branch Prediction and Intel® Hyper-Threading Technology (Intel® HT Technology)

In a processor supporting Intel® Hyper-Threading Technology, a core (or physical processor) may include multiple logical processors. On such processors, the logical processors sharing a core may share indirect branch predictors. As a result of this sharing, on processors without enhanced IBRS, software on one of a core's logical processors may be able to control the predicted target of an indirect branch executed on another logical processor on the same core. This behavior can be disabled using the Single Thread Indirect Branch Predictors (STIBP) control described below.

Such sharing occurs only within a core. Software executing on a logical processor of one core cannot control the predicted target of an indirect branch by a logical processor of a different core.

3.1.2 Indirect Branch Speculation Control Mechanisms

Intel has developed indirect branch predictor controls, which are interfaces between the processor and system software to manage the state of indirect branch predictors.

All supported Intel processors (when running with up-to-date microcode) provide three indirect branch control mechanisms:

- Indirect Branch Restricted Speculation (IBRS): Restricts indirect branch predictions, which can be used by virtual machine manager (VMM) or operating system code to prevent the use of predictions from another security domain. Recent processors support enhanced IBRS, which can be enabled once and never disabled (*always on* mode).

[3] This is an example of attacker-controlled jump redirection.

- Single Thread Indirect Branch Predictors (STIBP): Prevents indirect branch predictions from being controlled by a sibling hyperthread. Processors which support enhanced IBRS always have this behavior, regardless of the setting of STIBP.

- Indirect Branch Predictor Barrier (IBPB): Prevents indirect branch predictions after the barrier from being controlled by software executed before the barrier. IBPB also acts as a barrier for the Fast Store Forwarding Predictor and Data Dependent Prefetchers (refer to Section 4.1 Overview of Data Speculation), where relevant. This allows VMM and operating system code to provide isolation when switching between guests or userspace applications which execute in different security domains.

Some recent Intel processors also support additional indirect branch control mechanisms which focus on specific indirect branch predictors or behaviors. Some examples include the `IPRED_DIS_U`, `IPRED_DIS_S`, `RRSBA_DIS_U`, `RRSBA_DIS_S` and `BHI_DIS_S` bits in the IA32_SPEC_CTRL MSR.

System software can use these indirect branch control mechanisms to defend against branch target injection attacks.

3.1.2.1 Predictor Mode

Intel processors support different modes of operation corresponding to different levels of privilege. VMX root operation (for a virtual-machine monitor, or *host*) is more privileged than VMX non-root operation (for a virtual machine, or *guest*). Within either VMX root operation or VMX non-root operation, *supervisor mode* (CPL < 3) is more privileged than *user mode* (CPL= 3).

To prevent inter-mode attacks based on branch target injection, it is important to ensure that less privileged software cannot control the branch target prediction in more privileged software. For this reason, it is useful to introduce the concept of *predictor mode* associated with different modes of operation as mentioned above. There are four predictor modes: host-supervisor, host-user, guest-supervisor, and guest-user.

The guest predictor modes are considered less privileged than the host predictor modes. Similarly, the user predictor modes are considered less privileged than the supervisor predictor modes.

There are operations that may be used to transition between unrelated software components but do not change `CPL` or cause a `VMX` transition. These operations do not change predictor mode. Examples include `MOV` to `CR3`, `VMPTRLD`, EPTP switching (using VM function 0), and `GETSEC[SENTER]`.

3.1.2.2 Indirect Branch Restricted Speculation (IBRS)

Indirect branch restricted speculation (IBRS) is an indirect branch control mechanism that restricts speculation of indirect branches. A processor supports IBRS if it enumerates CPUID.(EAX=7H,ECX=0):EDX[26] as 1.

3.1.2.2.1 IBRS: Basic Support

Processors that support IBRS provide the following guarantees without any enabling by software:

- The predicted targets of near indirect branches executed in an enclave (a protected container defined by Intel® SGX) cannot be controlled by software executing outside the enclave.

- If the default treatment of system-management interrupts (SMIs) and system management mode SMM is active, software executed before a SMI cannot control the predicted targets of indirect branches executed in SMM after the SMI.
- The predicted targets of near indirect branches executed inside a Trust Domain (TD), a virtual machine managed by Intel® Trust Domain Extensions (Intel® TDX) module, cannot be controlled by software executing outside the TD.

3.1.2.2.2 IBRS: Support Based on Software Enabling

IBRS provides a method for critical software to protect their indirect branch predictions.

If software sets IA32_SPEC_CTRL.IBRS to 1 after a transition to a more privileged predictor mode, predicted targets of indirect branches executed in that predictor mode with IA32_SPEC_CTRL.IBRS = 1 cannot be controlled by software that was executed in a less privileged predictor mode[4]. Additionally, when IA32_SPEC_CTRL.IBRS is set to 1 on any logical processors of that core, the predicted targets of indirect branches cannot be controlled by software that executes (or has executed previously) on another logical processor of the same core. Therefore, it is not necessary to set bit 1 (STIBP) of the IA32_SPEC_CTRL MSR when IBRS is set to 1.

If IA32_SPEC_CTRL.IBRS is already 1 before a transition to a more privileged predictor mode, some processors may allow the predicted targets of indirect branches executed in that predictor mode to be controlled by software that executed before the transition. Software can avoid this by using `WRMSR` on the IA32_SPEC_CTRL MSR to set the IBRS bit to 1 after any such transition, regardless of the bit's previous value. It is not necessary to clear the bit first; writing it with a value of 1 after the transition suffices, regardless of the bit's original value.

Setting IA32_SPEC_CTRL.IBRS to 1 does not suffice to prevent the predicted target of a near return from using an RSB entry created in a less privileged predictor mode. Software can avoid this by using an RSB overwrite sequence[5] following a transition to a more privileged predictor mode. It is not necessary to use such a sequence following a transition from user mode to supervisor mode if supervisor-mode execution prevention (SMEP) is enabled. SMEP prevents execution of code on user mode pages, even speculatively, when in supervisor mode. User mode code can only insert its own return addresses into the RSB, not the return addresses of targets on supervisor mode code pages. On processors without SMEP where separate page tables are used for the OS and applications, the OS page tables can map user code as *no-execute*. The processor will not speculatively execute instructions from a translation marked no-execute.

Enabling IBRS does not prevent software from controlling the predicted targets of indirect branches of unrelated software executed later at the same predictor mode (for example, between two different user applications, or two different virtual machines). Such isolation can be ensured through use of IBPB, described in Section 3.1.2.4 Indirect Branch Predictor Barrier (IBPB).

[4] A transition to a more privileged predictor mode through an `INIT#` is an exception to this and may not be sufficient to prevent the predicted targets of indirect branches executed in the new predictor mode from being controlled by software operating in a less privileged predictor mode.

[5] An RSB overwrite sequence is a sequence of instructions that includes 32 more near `CALL` instructions with non-zero displacements than it has near `RET`s.

Enabling IBRS on one logical processor of a core with Intel HT Technology may affect branch prediction on other logical processors of the same core. For this reason, software should disable IBRS (by clearing IA32_SPEC_CTRL.IBRS) prior to entering a sleep state (for example, by executing `HLT` or `MWAIT`) and re-enable IBRS upon wakeup and prior to executing any indirect branch.

3.1.2.2.3 Enhanced IBRS

Some processors may enhance IBRS by simplifying software enabling and improving performance. A processor supports *enhanced IBRS* if `RDMSR` returns a value of 1 for bit 1 of the IA32_ARCH_CAPABILITIES MSR.

Enhanced IBRS supports an *always on* model in which IBRS is enabled once (by setting IA32_SPEC_CTRL.IBRS) and never disabled. If IA32_SPEC_CTRL.IBRS = 1 on a processor with enhanced IBRS, the predicted targets of indirect branches executed cannot be controlled by software executed in a less privileged predictor mode or on another logical processor.

As a result, software operating on a processor with enhanced IBRS need not use `WRMSR` to set IA32_SPEC_CTRL.IBRS after every transition to a more privileged predictor mode. Software can isolate predictor modes effectively simply by setting the bit once. Software need not disable enhanced IBRS prior to entering a sleep state such as `MWAIT` or `HLT`.

On processors with enhanced IBRS, an RSB overwrite sequence may not suffice to prevent the predicted target of a near return from using an RSB entry created in a less privileged predictor mode. Software can prevent this by enabling SMEP (for transitions from user mode to supervisor mode) and by having IA32_SPEC_CTRL.IBRS set during VM exits. Processors with enhanced IBRS still support the usage model where IBRS is set only in the OS/VMM for OSes that enable SMEP. To do this, such processors will manage guest behavior such that it cannot control the RSB after a VM exit once IBRS is set, even if IBRS was not set at the time of the VM exit. If the guest has cleared IBRS, the hypervisor should set IBRS after the VM exit, just as it would do on processors supporting IBRS but not enhanced IBRS. As with IBRS, enhanced IBRS does not prevent software from affecting the predicted target of an indirect branch executed at the same predictor mode. For such cases, software should use the IBPB command, described in Section 3.1.2.4 Indirect Branch Predictor Barrier (IBPB).

On processors with *enhanced* IBRS support, Intel recommends that IBRS be set to 1 and left set. The traditional IBRS model of setting IBRS only during ring 0 execution is just as secure on processors with enhanced IBRS support as it is on processors without enhanced IBRS, but the `WRMSR`s on ring transitions and/or VM exit/entry will cost performance compared to just leaving IBRS set. Again, there is no need to use STIBP when IBRS is set. However, IBPB should still be used when switching to a different application/guest that does not trust the last application/guest that ran on a particular hardware thread.

Guests in a VM migration pool that includes hardware without enhanced IBRS may not have IA32_ARCH_CAPABILITIES.IBRS_ALL (enhanced IBRS) enumerated to them, and thus may use the traditional IBRS usage model of setting IBRS only in ring 0. For performance reasons, once a guest has been shown to frequently write IA32_SPEC_CTRL, we do not recommend that the VMM cause a VM exit on such `WRMSR`s. The VMM running on processors that support enhanced IBRS should allow the IA32_SPEC_CTRL-writing guest to control guest IA32_SPEC_CTRL. The VMM should thus set IBRS after VM exits from such guests to protect itself (or use alternative techniques like retpoline, secret removal, or indirect branch removal).

On processors without *enhanced* IBRS, Intel recommends using retpoline or setting IBRS only during ring 0 and VMM modes. IBPB should be used when switching to a different process/guest that does not trust the last process/guest that ran on a particular hardware thread. For performance reasons, IBRS should not be left set during application execution.

3.1.2.3 Single Thread Indirect Branch Predictors (STIBP)

As noted in Section 3.1.1.1 Indirect Branch Prediction and Intel® Hyper-Threading Technology (Intel® HT Technology), the logical processors sharing a core may share indirect branch predictors, allowing one logical processor to control the predicted targets of indirect branches by another logical processor of the same core.

Single thread indirect branch predictors (STIBP) is an indirect branch control mechanism that restricts the sharing of indirect branch prediction between logical processors on a core. A processor supports STIBP if it enumerates CPUID.(EAX=7H,ECX=0):EDX[27] as 1. Setting bit 1 (STIBP) of the IA32_SPEC_CTRL MSR on a logical processor prevents the predicted targets of indirect branches on any logical processor of that core from being controlled by software that executes (or executed previously) on another logical processor of the same core.

Unlike IBRS and IBPB, STIBP does not affect all branch predictors that contain indirect branch predictions. STIBP only affects those branch predictors where software on one hardware thread can create a prediction that can then be used by the other hardware thread for indirect branches. This is part of what makes STIBP have lower performance overhead than IBRS on some implementations.

It is not necessary to use IBPB after setting STIBP in order to make the STIBP effective. STIBP provides isolation of indirect branch prediction between logical processors on the same core only when it is set. In particular, it is not a branch prediction barrier - setting and then unsetting STIBP does not prevent indirect branch predictions from being controlled by previously executed code and/or code on other logical processors.

Processes that are particularly security-sensitive may wish to have STIBP be set when they execute to prevent their indirect branch predictions from being controlled by another hardware thread on the same physical core. On some older Intel® Core-family processors, this comes at significant performance cost to both hardware threads due to disabling some indirect branch predictors (as described earlier). Because of this, we do not recommend that STIBP be set during all application execution on processors that do not support enhanced IBRS.

Indirect branch predictors are never shared across cores. Thus, the predicted target of an indirect branch executed on one core can never be affected by software operating on a different core. It is not necessary to set IA32_SPEC_CTRL.STIBP to isolate indirect branch predictions from software operating on other cores.

Many processors do not allow the predicted targets of indirect branches to be controlled by software operating on another logical processor, regardless of STIBP. These include processors on which Intel Hyper-Threading Technology is not enabled and those that do not share indirect branch predictor entries between logical processors. To simplify software enabling and enhance workload migration, STIBP may be enumerated (and setting IA32_SPEC_CTRL.STIBP allowed) on such processors.

A processor may enumerate support for the IA32_SPEC_CTRL MSR (by enumerating CPUID.(EAX=7H,ECX=0):EDX[26] as 1) but not for STIBP (CPUID.(EAX=7H,ECX=0):EDX[27] is enumerated as 0). On such processors, execution of `WRMSR` to IA32_SPEC_CTRL ignores the value of bit

1 (STIBP) and does not cause a general-protection exception (`#GP`) if bit 1 of the source operand is set. It is expected that this fact will simplify virtualization in some cases.

As noted in Section 3.1.2.1, Predictor Mode, Intel processors support different modes of operation corresponding to different levels of privilege. VMX root operation (for a virtual-machine monitor, or host) is more privileged than VMX non-root operation (for a virtual machine, or guest). Within either VMX root operation or VMX non-root operation, supervisor mode (CPL < 3) is more privileged than user mode (CPL= 3).

To prevent inter-mode attacks based on branch target injection, it is important to ensure that less privileged software cannot control the branch target prediction in more privileged software. For this reason, it is useful to introduce the concept of predictor mode associated with different modes of operation as mentioned above. There are four predictor modes: host-supervisor, host-user, guest-supervisor, and guest-user.

The guest predictor modes are considered less privileged than the host predictor modes. Similarly, the user predictor modes are considered less privileged than the supervisor predictor modes.

There are operations that may be used to transition between unrelated software components but do not change CPL or cause a VMX transition. These operations do not change predictor mode. Examples include `MOV` to `CR3`, `VMPTRLD`, EPTP switching (using VM function 0), and `GETSEC[SENTER]`.

Enabling IBRS prevents software operating on one logical processor from controlling the predicted targets of indirect branches executed on another logical processor. For that reason, it is not necessary to enable STIBP when IBRS is enabled.

Recent Intel processors, including all processors which support enhanced IBRS, provide this isolation for indirect branch predictions between logical processors without the need to set STIBP.

Enabling STIBP on one logical processor of a core with Intel Hyper-Threading Technology may affect branch prediction on other logical processors of the same core. For this reason, on processors which do not support enhanced IBRS, software should disable STIBP (by clearing IA32_SPEC_CTRL.STIBP) prior to entering a sleep state (for example, by executing `HLT` or `MWAIT`) and re-enable STIBP upon wakeup and prior to executing any indirect branch.

3.1.2.4 Indirect Branch Predictor Barrier (IBPB)

The *indirect branch predictor barrier* (IBPB) is an indirect branch control mechanism that establishes a barrier, preventing software that executed before the barrier from controlling the predicted targets of indirect branches[6] executed after the barrier on the same logical processor. A processor supports IBPB if it enumerates CPUID.(EAX=7H,ECX=0):EDX[26] as 1. IBPB can be used to help mitigate Branch Target Injection.

[6] Note that indirect branches include near call indirect, near jump indirect and near return instructions; as documented by the speculative execution side channel mitigations guidance. Because it includes near returns, it follows that RSB entries created before an IBPB command cannot control the predicted targets of returns executed after the command on the same logical processor.

The IBPB also provides other domain isolation properties regarding speculative execution, such as for the Fast Store Forwarding Predictor and Data Dependent Prefetchers where relevant.

Unlike IBRS and STIBP, IBPB does not define a new mode of processor operation that controls the branch predictors. As a result, it is not enabled by setting a bit in the IA32_SPEC_CTRL MSR. Instead, IBPB is an operation that software executes when necessary.

Software executes an IBPB command by writing the IA32_PRED_CMD MSR to set bit 0 (IBPB). This can be done either using the `WRMSR` instruction or as part of a VMX transition that loads the MSR from an MSR-load area. Software that executed before the IBPB command cannot control the predicted targets of indirect branches executed after the command on the same logical processor. The IA32_PRED_CMD MSR is write-only, and it is not necessary to clear the IBPB bit before writing it with a value of 1.

IBPB should be used when switching to a different application/guest that does not want its indirect branch predictions to be controlled by previous applications/guests that ran on that logical processor in the same predictor mode.

IBPB can be used in conjunction with IBRS to account for cases that IBRS does not cover:

- As noted in Section 3.1.2.2 Indirect Branch Restricted Speculation (IBRS), IBRS does not prevent software from controlling the predicted target of an indirect branch of unrelated software (for example, a different user application or a different virtual machine) executed at the same predictor mode. Software can aim to prevent such control by executing an IBPB command when changing the identity of software operating at a particular predictor mode (for example, when changing user applications or virtual machines).
- Software may choose to clear IA32_SPEC_CTRL.IBRS in certain situations (for example, for execution with CPL = 3 in VMX root operation). In such cases, software can use an IBPB command on certain transitions (for example, after running an untrusted virtual machine) to prevent software that executed earlier from controlling the predicted targets of indirect branches executed subsequently with IBRS disabled.

Note that, on some processors that do not enumerate `PBRSB_NO`, there is an exception to the IBPB-established barrier for RSB-based predictions. On these processors, a `RET` instruction that follows VM exit or IBPB without a corresponding `CALL` instruction may use the linear address following the most recent `CALL` instruction executed prior to the VM exit or IBPB as the RSB prediction (refer to the Post-barrier Return Stack Buffer Predictions guidance). In these cases, software can use special code sequences (refer to Section 3.1.3.1 Return Stack Buffer stuffing to steer RSB predictions to benign code regions that restrict speculation.

3.1.2.5 Other Indirect Branch Predictor Controls

The `BHI_DIS_S` indirect predictor control prevents predicted targets of indirect branches executed in CPL< 3 from being selected based on branch history from branches executed in CPL3. While set in the VMX root (host), it also prevents predicted targets executed in CPL0 (ring 0/root) from being selected based on branch history from branches executed in a VMX non-root (guest). It may not prevent predicted targets executed in CPL3 of VMX root from being based on branch history for branches executed in a VMX non-root (guest).

Enumeration of `BHI_NO` indicates that, even when `BHI_DIS_S` is not set, the processor prevents predicted targets of indirect branches executed in CPL < 3 from being selected based on branch history from branches executed in CPL3, other than RSB-based return predictions. Processors which enumerate `BHI_NO` also always prevent predicted targets executed in VMX root from being selected based on branch history from branches executed in VMX non-root (guest)

On processors that do not enumerate `BHI_NO`, the history of executed branches before IBPB can influence new indirect branch predictions following IBPB through the branch history buffer (BHB). Intel is not aware of any production code where this behavior allows a transient execution attack, since IBPB isolates indirect branch targets.

The `IPRED_DIS_U` (affecting CPL3) and `IPRED_DIS_S` (affecting CPL < 3) controls, when active, prevent transient execution at predicted targets of an indirect near `JMP`/`CALL` before the target is resolved[7]. This includes transient execution at past targets of that same branch. Transient execution at predicted targets of a near `RET` prediction will only occur for RSB-based return predictions, or for linear address 0. Note that, as previously documented, fall-through speculation to instruction bytes following an indirect `JMP`/`CALL` or speculation to linear address 0 may still occur.

When the `RRSBA_DIS_S` (affecting CPL < 3) and `RRSBA_DIS_U` (affecting CPL3) indirect predictor controls are set, transient execution at predicted targets of a near `RET` prediction will only occur for RSB-based return predictions, or for linear address 0.

3.1.3 Software Techniques for Indirect Speculation Control

Besides the hardware-based mechanisms described above, software mechanisms can also be used to limit indirect branch speculation where hardware mechanisms are not available.

For example, indirect branch prediction can be suppressed in some cases by using a software-based approach called *retpoline*, which was developed by Google*. Details of retpoline are described in Retpoline: A Branch Target Injection Mitigation.

3.1.3.1 Return Stack Buffer stuffing

RSB stuffing (also known as an *RSB overwrite sequence*) is a software technique to fill the RSB with trusted-software-controlled return targets.

3.1.3.1.1 RSB stuffing on VM exit

To avoid guest control of the predicted targets of VMM RET instructions after a VM exit, on processors without enhanced IBRS support, a VMM can apply RSB stuffing with 32 return targets. RSB stuffing after an VM exit is not required on processors which support enhanced IBRS, as described in Section 3.1.2.2.3 Enhanced IBRS.

3.1.3.1.2 RSB stuffing to avoid underflow

Under some circumstances on processors that have RSBA or RRSBA behavior, situations such as a deep call stack or imbalanced `CALL` and `RET` instructions may result in RSB underflowing and alternate

[7] Note that this does not cover attacker-controlled jump redirection, which is described in volume 1, section 17.1.3 “Speculative Behavior when CET is Enabled” of the Intel Software Developers Manual.

predictors being used to predict the return address of `RET` instructions. The properties of both IBRS and enhanced IBRS continue to apply to such predictions.

On processors with RSBA behavior where software is using retpoline rather than IBRS, software may wish to apply RSB stuffing to avoid RSB underflow in some cases, as described in Retpoline: A Branch Target Injection Mitigation.

3.1.3.2 Branch History Buffer Control

To address Branch History Injection, software can use a code sequence to control speculation that arises from collisions in the Branch History Buffer (BHB). This code sequence overwrites the branch history after domain transitions to prevent the previous domain from influencing BHB-based indirect branch prediction in the current domain. As microarchitectural details of the BHB may change in future processors, Intel recommends using hardware-based controls, such as, `BHI_DIS_S`, where available.

3.1.4 Guidance for Managing Indirect Branches

Software can effectively restrict speculation and protect against speculation-based attacks in use cases that have an increased risk of exploitation. This includes mechanisms to avoid speculation in more privileged modes being manipulated by less privileged code to reveal sensitive information. It also includes mechanisms to avoid such manipulation between applications or between virtual machines.

3.1.4.1 Operating Systems

Intel recommends operating systems enable enhanced IBRS (eIBRS) to prevent ring 3 software from influencing predictions and speculation in kernel. On older processors, where enhanced IBRS is not available, the operating system should use IBRS instead. In addition, operating systems should disable unprivileged extended Berkley Packet Filter (eBPF) to prevent unprivileged users from introducing potentially harmful code into the kernel domain and misusing it via branch history injection (BHI). Operating systems should also keep Supervisor Mode Execution Prevention (SMEP) enabled to prevent the kernel from executing and speculating on code from untrusted userspace.

3.1.4.2 VMM

Intel recommends VMMs enable enhanced IBRS to restrict speculation and prevent accidental data leakage. VMMs running on processors that support enhanced IBRS should allow the guest to use IA32_SPEC_CTRL and efficiently control the mitigations itself. The VMM should thus set IBRS after VM exits from such guests to protect itself. If eIBRS is not available on a processor, VMMs should mitigate Branch Target Injection with other techniques to isolate the VMM against its guests. VMMs may also wish to apply IBPB when switching between guests, prior to VM entry to the new guest; see Section 3.1.2.4 Indirect Branch Predictor Barrier (IBPB). See also Section 3.1.3.1.1 (RSB stuffing on VM exit).

VMMs may also wish to consider mitigating specific vulnerabilities; see the specific guidance for more details:

- On processors affected by Post-Barrier Return Stack Buffer Predictions, the VMM may need to execute an additional software sequence on VM exit.

- On some processors affected by Indirect Target Selection, any VMM relying on enhanced IBRS's guest/host isolation may need to apply additional software mitigations to mitigate Branch Target Injection attacks.

3.1.4.3 Disclosure Gadgets in Software

Intel recommends adapting software, such as by using bounds clipping or `LFENCE`, to fix potentially exploitable gadgets as they are found. Refer to section 2.2 Restricting Speculative Execution and section 3.2.4 Software Techniques for Conditional Speculation Control for more information.

3.1.4.4 System Management Mode (SMM)

On certain processors from the Skylake generation, System Management Interrupt (SMI) handlers can leave the RSB in a state that OS code does not expect. To avoid RSB underflow on return from SMI and ensure retpoline implementations in the OS and VMM work properly, on these processors, an SMI handler may implement RSB stuffing before returning from System Management Mode (SMM).

3.2 Conditional Branches

Intel processors use conditional branch predictors to predict the direction of conditional branch instructions before their actual execution. This allows the processor to fetch and speculatively execute instructions on the predicted execution path after the conditional branch. Speculative execution side channels (aka Transient Execution Attacks) that are based around conditional branch prediction are classified as Spectre Variant 1.

3.2.1 Overview of Bounds Check Bypass

Bounds check bypass is a side channel method that takes advantage of the speculative execution that may occur following a conditional branch instruction. Specifically, the method is used in situations in which the processor is checking whether an input is in bounds (for example, while checking whether the index of an array element being read is within acceptable values). The processor may issue operations speculatively before the bounds check resolves. If the attacker contrives for these operations to access out-of-bounds memory, information may be inferred by the attacker in certain circumstances.

3.2.1.1 Bounds Check Bypass Store

One subvariant of this technique, known as bounds check bypass store, is to use speculative stores to overwrite younger speculative loads in a way that creates a side channel controlled by a malicious actor.

Refer to the example *bounds check bypass store* sequence below:

```
int function(unsigned bound, unsigned long user_key) {

        unsigned long data[8];
        /* bound is trusted and is never more than 8 */
        for (int i = 0; i < bound; i++){
              data[i] = user_key;
        }

        return 0;
}
```

The example above does not by itself allow a bounds check bypass attack. However, it does allow the attack to speculatively modify memory, and therefore could potentially be used to chain attacks. For example, it is possible that the above sequence might speculatively overwrite the return address on the stack with `user_key`. This may allow a malicious actor to specify a `user_key` that is actually the instruction pointer of a disclosure gadget that they wish to be speculatively executed.

The steps below describe how an example attack using this method might occur:

1. The CPU conditional branch predictor predicts that the loop will iterate 10 iterations, when in reality the loop should have only executed 8 times. After the 10th iteration, the predictor will resolve, fall through, and execute the following instructions. However, the 9th iteration of the loop may speculatively overwrite the return address on the stack.

2. The CPU decodes the `RET` and speculatively fetches instructions based on the prediction in the return stack buffer (RSB). The CPU may speculatively execute those instructions.

3. `RET` loads the value that it believes is at the top of the stack (but which came from the speculative store of `user_key` in step 1) and redirects the instruction pointer to that value. The results of any operations speculatively executed in step 2 are discarded.

4. The disclosure gadget at the instruction pointer of `user_key` (which was specified by the malicious actor) speculatively executes and creates a side channel that can be used to reveal data specified by the malicious actor.

5. The conditional jump that should have ended the loop then executes and redirects the instruction pointer to the next instruction after the loop. This discards the speculative store of `user_key` that overwrote the return address on the stack, as well as all other operations between step 1 and step 4.

6. The CPU executes the `RET` again, and the program continues.

Where the compiler has spilled variables to the stack, the store can also be used to target those spilled values and speculatively modify them to enable another attack to follow. An example of this would be by targeting the base address of an array dereference or the limit value.

SMEP will prevent the attack described above from causing a supervisor `RET` to speculatively execute code in user mode pages. Intel® Control flow Enforcement Technology (Intel® CET) can also help prevent speculative execution of instructions at incorrect indirect branch targets.

This example can be mitigated either by applying `LFENCE` before the `RET` (after the loop ends), by using bounds clipping to ensure that store operations do not occur outside of the array's bounds, even speculatively, or by ensuring that incorrect return pointer is detected and that the return does not speculatively use the incorrect value.

A second variant of this method can occur where a user value is being copied into an array, either on the stack or adjacent to function pointers. As discussed previously, the processor may speculatively execute a loop more times than is actually needed. If this loop moves through memory writing malicious actor-controlled values, then the malicious actor may be able to speculatively perform a buffer overrun attack.

```
int filltable(uint16_t *from)

{

      uint16_t buffer[64];

      int i;


      for (i = 0; i < 64; i++)

            buffer[i] = *from++;

}
```

In some cases, the example above might speculatively copy more bytes than 64 into the array, changing the return address speculatively used by the processor so that it instead returns to a user-controlled gadget.

As the execution is speculative, some processors will allow speculative writes to read-only memory and will reuse that data speculatively. Therefore, while placing function pointers into write-protected space is a good general security mitigation, doing so is not sufficient mitigation in this case.

3.2.2 Identifying Bounds Check Bypass Vulnerabilities

The following section examines common instances of bounds check bypass, including the *bounds check bypass store* variant, but should not be considered a comprehensive list. It describes how to analyze potential *bounds check bypass* and *bounds check bypass store* vulnerabilities found by static analysis tools or manual code inspection and presents mitigation techniques that may be used. This document does not include any actual code from any real product or open source release, nor does it discuss or recommend any specific analysis tools.

3.2.2.1 Common Attributes for Bounds Check Bypass Vulnerabilities

Bounds check bypass code sequences have some common features: they generally operate on data that is controlled or influenced by a malicious actor, and they all have some kind of side-effect that can be *observed* by the malicious actor. In addition, the processor's speculative execution sequence executes in a way which would be thrown away in a normally retired execution sequence. In bounds check bypass store variants, data is speculatively written at locations that would be out of bounds under normal execution. That data is later speculatively used to execute code and cause observable side-effects, creating a side channel.

3.2.2.2 Loads and Stores

A vulnerable code fragment forming a disclosure gadget is made up of two elements. The first is an array or pointer dereference that depends upon an untrusted value, for example, a value from a potentially malicious application. The second element is usually a load or store to an address that is dependent upon the value loaded by the first element. Refer to Microsoft*'s blog for further details.

As bounds check bypass is based upon speculation, code can be vulnerable even if that untrusted value is correctly tested for bounds before use.

The classic general example of such a sequence in C is:

```
if (user_value >= 0 && user_value < LIMIT) {

       x = table[user_value];

       node = entry[x];

} else

       return ERROR;
```

For such a code sequence to be vulnerable, both elements must be present. Furthermore, the untrusted value must be under the malicious actor's control.

When the code executes, the processor has to decide if the `user_value < LIMIT` conditional is true or false. It remembers the processor register state at this point and speculates (makes a guess) that `user_value` is below `LIMIT` and begins executing instructions as if this were true. Once the processor realizes it guessed incorrectly, it throws away the computation and returns an error. The attack relies upon the fact that before it realizes the guess was incorrect, the processor has read both `table[user_value]`, pointing into memory beyond the intended limit, and has read `entry[x]`. When the processor reads `entry[x]`, it may bring in the corresponding cache line from memory into the L1 cache. Later, the malicious actor can time accesses to this address to determine whether the corresponding cache line is in the L1 data cache. The malicious actor can use this timing to discover the value `x`, which was loaded from a malicious actor-specified location.

The two components that make up this vulnerable code sequence can be stretched out over a considerable distance and through multiple layers of function calls. The processor can speculatively execute many instructions—a number sufficient to pass between functions, compilation units, or even software exception handlers such as `longjmp` or `throw`. The processor may speculate through locked operations, and use of `volatile` will not change the vulnerability of the code being exploited.

There are several other sequences that may be used to infer information. Anything that tests some property of a value and loads or stores according to the result may leak information. Depending upon the location of `foo` and `bar`, the example below might be able to leak bit 0 of arbitrary data.

```
if (user_value >= LIMIT)
      return ERROR;
x = table[user_value];
if (x & 1)
      foo++;
else
      bar++;
```

When evaluating code sequences for vulnerability to bounds check bypass, the critical question is whether different behavior could be observed as a property of `x`.

This question can be very challenging to answer from code inspection, especially when looking for any specific code pattern. For instance, if a value is passed to a function call, then that function call must be inspected to ensure it does not create any observable interactions. Consider the following example:

```
if (user_value >= LIMIT)
      return ERROR;
x = lengths[user_value];
if (x)
      memset(buffer, 0, 64 * x);
```

Here, `x` influences how much memory is cleared by `memset`() and might allow the malicious actor to discern something about the value of x from which cache lines the speculatively executed `memset` touches.

Remember that conditional execution is not just `if`, but may also include `for` and `while` as well as the C ternary (`?:`) operator and situations where one of the values is used to index an array of function pointers.

3.2.2.3 Typecasting and Indirect Calls

Typecasting can be a problematic area to analyze and often conceals real examples that can be exploited. This is especially challenging in C++ because you are more likely to have function pointers embedded in objects and overloaded operators that might behave in type-dependent fashion.

Two classes of typecasting problems are relevant to bounds check bypass attacks:

1. *Code/data mismatches*. Speculation causes "class `Foo`" code to be speculatively executed on "class `Bar`" data using gadgets supplied with `Foo` to leak information about `Bar`.
2. *The type confusion is combined with some observable effect, like the load/store effects discussed above*. For example, if `Foo` and `Bar` are different sizes, a malicious actor might be able to learn something about memory past the end of `objects[]` using something like the example below.

```
        type = objects[index];
        if (index >= len)
              return -EINVAL;
        if (type == TYPE_FOO)
              memset(ptr, 0, sizeof(Foo));
        else
              memset(ptr, 0, sizeof(Bar));
```

Take care when considering any code where a typecast occurs based upon a speculated value. The processor might guess the type incorrectly and speculatively execute instructions based on that incorrect type. Newer processors that enable Intel® OS Guard, also known as Supervisor-Mode Execution Prevention (SMEP), will prevent ring 0 code from speculatively executing ring 3 code. All major operating systems (OSes) enable SMEP support by default if the hardware supports it. Older processors, however, might speculate the type incorrectly, load data that the processor thinks are function pointers, or speculate into lower addresses that might be directly controlled by a malicious actor.

For example:

```
      if (flag & 4)
            (Foo *)ptr->process(x);
      else
            (Bar *)ptr->process(x);
```

If the `Foo` and `Bar` objects are different and have different memory layouts, then the processor will speculatively fetch a pointer offset of `ptr` and branch to it.

Consider the following example:

```
      int call; /* from user */
      if (call >= 0 && call < MAX_FUNCTION)
                  function_table[call](a,b,c);
```

On first analysis this code might seem safe. We reference `function_table[call]`, but `call` is the user's own, known value. However, during speculative execution, the processor might incorrectly speculate through the `if` statement and speculatively execute invalid addresses. Some of these addresses might be mapped to user pages in memory or might contain values that match suitable gadgets for ROP attacks.

A less obvious variant of this case is `switch` statements. Many compilers will convert some classes of switch statement into jump tables. Refer to the following example code:

```
switch(x) {

case 0: return y;

case 1: return z;

...

default: return -1;

}
```

Code similar to this will often be implemented by the compiler as shown:

```
if (x < 0 || x > 2) return -1;

goto  case[x];
```

Therefore, when using `switch()` with an untrusted input, it might be appropriate to place an `lfence` before the switch so that `x` has been fully resolved before the implicit bounds check.

3.2.2.4 Speculative Loops

A final case to consider is loops that speculatively overrun. Consider the following example:

```
while (++x < limit) {

      y = u[x];

      thing(y);

}
```

The processor will speculate the loop condition, and often speculatively execute the next iteration of the loop. This is usually fine, but if the loop contains code that reveals the contents of data, then you might need to apply mitigations to avoid exposing data beyond the intended location of the loop. This means that even if the loop limit is properly protected before the processor enters the loop, unless the loop itself is protected, the loop might leak a small amount of data beyond the intended buffer on the speculative path.

3.2.2.5 Disclosure Gadgets

In addition to the load and store disclosure gadget referenced above, there may be additional gadgets based on the microarchitectural state. For example, using certain functional blocks, such as Intel® Advanced Vector Extensions (Intel® AVX), during speculative execution may affect the time it takes to subsequently use the block due to factors like the time required to power-up the block. Malicious actors can use a disclosure primitive to measure the time it takes to use the block. An example of such a gadget is shown below:

```
if (x > sizeof(table))

      return ERROR;

If (a[x].op == OP_VECTOR)

      avx_operation(a[x]);

else

      integer_operation(a[x]);
```

3.2.3 Conditional Branch Speculation Analysis

Controlling conditional branch speculation, such as bounds check bypass, is not generally relevant if your code doesn't have secrets that the user shouldn't be able to access. For example, a simple image viewer probably contains no meaningful secrets that should be inaccessible to software it interacts with. The user of the software could potentially use bounds check bypass attacks to access the image, but they could also just hit the save button.

On the other hand, an image viewer with support for secure, encrypted content with access authorized from a central system might need to care about bounds check bypass because a user may not be allowed to save the document in normal ways. While the user can't save such an image, they can trivially photograph the image and send the photo to someone, so protecting the image may be less important. However, any keys are likely to be far more sensitive.

There are also clear cases like operating system kernels, firmware (refer to the Host Firmware Speculative Execution Side Channel Mitigation technical paper) and managed runtimes (for example, Javascript* in web browsers) where there is both a significant interaction surface between differently trusted code, and there are secrets to protect.

Whether to apply mitigations, and what areas to target has to be part of your general security analysis and risk modelling, along with conventional security techniques, and resistance if appropriate to timing and other non-speculative side channel attacks. Bounds check bypass mitigations have performance impacts, so they should only be used where appropriate.

3.2.4 Software Techniques for Conditional Speculation Control

The recommended approach to mitigate bounds check bypass (Spectre v1) is to use bounds clipping, or `CMOVcc` in cases where bounds clipping is not appropriate. Bounds clipping is an effective mitigation in constraining speculative execution to prevent a side channel with data dependency on out-of-bounds array accesses. Alternative options, such as `LFENCE`, are described below.

3.2.4.1 Bounds Clipping

Software can use instructions, such as `CMOVcc`, `AND`, `ADC`, `SBB`, and `SETcc`, to constrain speculative execution and prevent bounds check bypass. This approach can avoid stalling the pipeline as `LFENCE` does. `CMOVcc`/`SETcc` will not introduce new forms of speculation (for example, they will not predict its direction). However, note that other speculation features may affect these instructions (for example, memory reads may be affected by memory disambiguation).

A simple example in C code:

```
unsigned int user_value;

if (user_value > 255)
      return ERROR;
x = table[user_value];
```

Can be made safe by instead using the following logic in array indexing:

```
volatile unsigned int user_value;

if (user_value > 255)
      return ERROR;
x = table[user_value & 255];
```

This works for powers of two array lengths or bounds only. In the example above the table array length is 256 (2^8), and the valid index should be <= 255. Take care that the compiler used does not optimize away the `& 255` operation. For other ranges, it’s possible to use `CMOVcc`, `ADC`, `SBB`, `SETcc`, and similar instructions to do verification.

3.2.4.2 CMOVcc

The `CMOVcc` instruction conditionally moves data based on condition codes set by a prior comparison. Unlike branch instructions, `CMOVcc` does not speculatively predict whether the move will be performed; it waits until the condition is resolved. This behavior means that `CMOVcc` is sufficient to mitigate bounds check bypass vulnerabilities even in cases where bounds clipping is not applicable.

The following example demonstrates how to define an inline function wrapper around the `CMOVNZ` (conditional move if not zero) instruction. This wrapper selects between one of two values (`if_true` and `if_false`) based on a Boolean condition (`cond`), without using a branch:

```
static inline uint64_t nospec_select(uint64_t if_true, uint64_t if_false,
uint8_t cond) {
    uint64_t result = if_false;

    // GCC/Clang inline assembly syntax
    __asm__ __volatile__(
        "test %2, %2\n\t"
        "cmovnz %1, %0\n\t"
        : "+r" (result)
        : "r" (if_true), "r" (cond)
        : "cc"
    );

    return result;
}
```

This mitigation technique can be applied to mitigate other kinds of vulnerabilities exposed by conditional branch speculation. The following example demonstrates using `CMOVcc` to mitigate a speculative type confusion gadget involving a tagged union. In dynamically typed systems, a value might represent either an integer or a pointer depending on a tag. Without mitigation, speculative execution might misinterpret the type and dereference an integer as a pointer:

```
#define TYPE_INT    0
#define TYPE_PTR    1

typedef struct {
    uint8_t type_tag;
    uint64_t value;
} tagged_value_t;

extern uint8_t secret_data[4096];

void process_tagged_value(tagged_value_t tv) {
    bool is_pointer = (tv.type_tag == TYPE_PTR);

    // Use CMOVcc to constrain the value: if not a pointer, use NULL
    uint64_t safe_ptr = nospec_select(tv.value, (uint64_t)NULL, is_pointer);

    // Dereference only if type check passed architecturally
    if (is_pointer) {
        uint8_t* ptr = (uint8_t*)safe_ptr;
        if (ptr != NULL) {
            uint8_t data = *ptr;
```

```
                // Process data...
            }
        } else {  // tv.type_tag == TYPE_INT
            // Handle as integer
            uint64_t integer_value = tv.value;
            // Process integer...
        }
    }
```

In this example, if the processor speculatively mispredicts that `tv.type_tag` equals `TYPE_PTR` when it actually equals `TYPE_INT`, the `CMOV` instruction ensures that `safe_ptr` is set to `NULL` during transient execution. This prevents the speculative dereference from accessing memory at an attacker-controlled address derived from the integer value, thereby blocking the disclosure gadget.

The next example uses a `CMOVcc` wrapper to mitigate a bounds-check bypass gadget.

```
    // Perform bounds check
    uint8_t in_bounds = (user_value >= 0 && user_value < LIMIT);

   // Use CMOVcc to constrain the index: if out of bounds, use 0 instead
    safe_index = nospec_select((uint64_t)user_value, 0, in_bounds);

    // Access array with constrained index
    x = array[safe_index];

    // nospec_select() prevents this operation from leaking x's value
    y = dependent_operation[x];

    // Return error if bounds check failed architecturally
    if (!in_bounds) {
        return ERROR;
    }
```

The `CMOVcc` mitigation prevents the speculative execution from using an out-of-bounds `user_value` to access an array, avoiding leaking secrets through dependent operations.

3.2.4.3 LFENCE

Where software desires to ensure that a branch is not incorrectly predicted under any circumstance, use of the above low-overhead Bounds Check Bypass mitigations may not suffice. For example, `CMOVcc` may speculatively execute based on a condition code predicted by other speculation features (for example, memory reads may be affected by memory disambiguation).

As a last resort in cases where `CMOVcc` and the other techniques mentioned above cannot be applied, software can use the `LFENCE` instruction, which will mitigate bounds check bypass vulnerabilities as well as ensuring that other speculation which may affect the branch direction has been resolved. The `LFENCE` instruction does not execute until all prior instructions have completed locally, and no later instruction begins execution until `LFENCE` completes. This means that `LFENCE` can have a significantly higher performance impact than other mitigations. Most vulnerabilities identified in section 3.2.2 Identifying Bounds Check Bypass Vulnerabilities can be protected by inserting an `LFENCE` instruction; for example:

```
if (user_value >= LIMIT)
      return ERROR;
lfence();
x = table[user_value];
node = entry[x];
```

Where `lfence()` is a compiler intrinsic or assembler inline that issues an `LFENCE` instruction and also tells the compiler that memory references may not be moved across that boundary. The `LFENCE` ensures that the loads do not occur until the condition has actually been checked. The memory barrier prevents the compiler from reordering references around the `LFENCE`, and thus breaking the protection.

3.2.4.3.1 Placement of LFENCE

To protect against speculative timing attacks, place the `LFENCE` instruction after the range check and branch, before any code that consumes the checked value, and before the data can be used in a gadget that might allow measurement.

For example:

```
if (x > sizeof(table))
      return ERROR;
lfence();
If (a[x].op == OP_VECTOR)
      avx_operation(a[x]);
else
      integer_operation(a[x]);
```

Unless there are specific reasons otherwise, and the code has been carefully analyzed, Intel recommends that the `lfence` is always placed after the range check and before the range checked value is consumed by other code, particularly if the code involves conditional branches.

3.2.4.3.2 Interaction with Memory Disambiguation

Memory disambiguation (as described in section 4.1 Overview of Data Speculation) can theoretically impact bounds clipping techniques when they involve a load from memory. In the following example, a `CMOVG` instruction is inserted to prevent a side channel from being created with data from any locations beyond the array bounds.

```
CMP RDX, [array_bounds]

JG out_of_bounds_input

MOV RCX, 0

MOV RAX, [RDX + 0x400000]

CMOVG RAX, RCX

<Further code that causes cache movement based on RAX value>
```

As an example, assume the value at `array_bounds` is 0x20, but that value was only just stored to `array_bounds` and that the prior value at `array_bounds` was significantly higher, such as 0xFFFF. The processor can speculatively execute the `CMP` instruction using a value of 0xFFFF for the loaded value due to the memory disambiguation mechanism. The instruction will eventually be re-executed with the intended `array_bounds` value of 0x20. This can theoretically cause the above sequence to support the creation of a side channel that reveals information about the memory at addresses up to 0xFFFF instead of constraining it to addresses below 0x20.

3.2.4.4 Multiple Branches

When using mitigations, particularly the bounds clipping mitigations, it is important to remember that the processor will speculate through multiple branches. Thus, the following code is not safe:

```
int *key;

int valid = 0;


if (input < NUM_ENTRIES) {

      lfence();

      key = &table[input];

      valid = 1;

}

….

if (valid)

      *key = data;
```

In this example, although the mitigation is applied correctly when the processor speculates that the first condition is valid, no protection is applied if the processor takes the out-of-range value and then speculates that `valid` is true on the other path. In this case it will probably expose the contents of a random register, although not in an easy to measure fashion.

Preinitializing `key` to `NULL` or another safe address will also not reliably work, as the compiler can eliminate the `NULL` assignment because it can never be used non-speculatively. In such cases it may be more appropriate to merge the two conditional code sections and put the code between them into a separate function that is called on both paths. Or you could add `volatile` to `key` and assign it to `NULL`—forcing the assignment to occur with `volatile`, or to add `lfence` before the final assignment.

3.2.4.5 Compiler-Based Approaches

Note that there are also compiler-based approaches that automatically augment software with instructions to constrain speculation and can help prevent Bounds Check Bypass, such as Speculative Load Hardening (clang) and the /Qspectre option (MSVC).

Compiler protections against buffer overwrites of return addresses, such as stack canaries, also provide some resistance to speculative buffer overruns. In situations where a loop speculatively overwrites the return address it will also speculatively trigger the stack protection diverting the speculative flow. However, stack canaries alone are not sufficient to protect from bounds check bypass attacks.

3.2.4.5.1 Microsoft Visual Studio* 2017 mitigations

The Microsoft Visual Studio* 2017 Visual C++ compiler toolchain includes support for the `/Qspectre` flag, which may automatically add mitigation for some bounds check bypass vulnerabilities. For more information and usage guidelines, refer to Microsoft's public blog and the Visual C++ /Qspectre option page for further details.

3.2.4.5.2 LFENCE in Intel® Fortran Compiler

You can insert an `LFENCE` instruction in Fortran applications as shown in the example below. Implement the following subroutine, which calls `_mm_lfence()` intrinsics:

```
      interface

              subroutine for_lfence() bind (C, name = "_mm_lfence")

                  !DIR$ attributes known_intrinsic, default :: for_lfence

              end subroutine for_lfence

          end interface


          if (untrusted_index_from_user .le. iarr1%length) then

              call for_lfence()

              ival = iarr1%data(untrusted_index_from_user)

              index2 = (IAND(ival,1)*z'100') + z'200'

              if(index2 .le. iarr2%length)

                  ival2 = iarr2%data(index2)
```

```
        endif
```

The `LFENCE` intrinsic is supported in the following Intel compilers:

- Intel® C++ Compiler 8.0 and later for Windows* OS, Linux* kernel, and macOS*.
- Intel® Fortran Compiler 14.0 and later for Windows, Linux, and macOS.

3.2.4.5.3 Compiler-driven Automatic Mitigations

Across the industry, there is interest in mitigations for bounds check bypass vulnerabilities that are provided automatically by compilers. Developers are continuing to evaluate the efficacy, reliability, and robustness of these mitigations and to determine whether they are best used in combination with, or in lieu of, the more explicit mitigations discussed above.

3.2.5 Operating System Mitigations

Where possible, dedicated operating system programming APIs should be used to mitigate bounds check bypass instead of using open-coded mitigations. Using the OS-provided APIs will help ensure that code can take advantage of new mitigation techniques or optimizations as they become available.

3.2.5.1 Linux* Kernel

The current Linux* kernel mitigation approach to bounds check bypass is described in the *speculation* file in the Linux kernel documentation. This file is subject to change as developers and multiple processor vendors determine their preferred approaches.

`barrier_nospec()`: on x86 architecture, this issues an `LFENCE` and provides the compiler with the needed memory barriers to perform the mitigation. It can be used as `barrier_nospec()`, as in the examples above. On non-Intel processors, `barrier_nospec()` either generates the correct barrier code for that processor, or does nothing if the processor does not speculate.

`array_index_nospec(index, size)`: this is an inline that, irrespective of the processor, provides a method to safely dereference an array element. Additionally, it returns `NULL` if the lookup is invalid. This allows you to take the many cases where you range check and then check that an entry is present, and fold those cases into a single conditional test.

Thus, we can turn:

```
if (handle < 32) {

      x = handle_table[handle];

      if (x) {

            function(x);

            return 0;

      }

}
```

```
        return -EINVAL;
```

Into:

```
        x = array_index_nospec( handle, 32);
        if (x == NULL)
              return -EINVAL;
        function(*x);
        return 0;
```

3.2.5.2 Microsoft Windows* OS

Windows C/C++ developers have a variety of options to assist in mitigating bounds check bypass (Spectre variant 1). The best option will depend on the compiler/code generation toolchains you are using. Mitigation options include manual and compiler assisted.

In mixed-mode compiler environments, where object files for the same project are built with different toolchains, there are varying degrees of mitigation options available. Developers need to be aware of and apply the appropriate mitigations depending on their code composition and appropriate toolchain support dependencies.

3.2.5.2.1 Inline/external assembly

The Intel® C Compiler and Intel® C++ Compiler provide inline assembly support for 32- and 64-bit targets, whereas Microsoft Visual* C++ only provides inline assembly support for 32-bit targets. Microsoft Macro Assembler* (MASM) or other external, third-party assemblers may also be used to insert `LFENCE` in assembly code.

3.2.5.2.2 _mm_lfence() compiler intrinsic

The Intel C Compiler, the Intel C++ Compiler, and the Microsoft Visual C++ compiler all support generating `LFENCE` instructions for 32- and 64-bit targets using the `_mm_lfence()` intrinsic. Other commodity compilers are likely to support a similar intrinsic.

The easiest way for Windows developers to gain access to the intrinsic is by including the *intrin.h* header file that is provided by the compilers. Some Windows SDK/WDK headers (for example, *winnt.h* and *wdm.h*) define the `_mm_lfence()` intrinsic to avoid inclusion of the compiler *intrin.h*. It is possible that you already have code that locally defines `_mm_lfence()` as well, or uses an already existing definition for the intrinsic.

3.2.5.2.3 LFENCE in C/C++

You can insert LFENCE instructions in a C/C++ program as shown in the example below:

```
#include <intrin.h>

#pragma intrinsic(_mm_lfence)


    if (user_value >= LIMIT)

    {

        return STATUS_INSUFFICIENT_RESOURCES;

    }

    else

    {

        _mm_lfence();   /* manually inserted by developer */

        x = table[user_value];

        node = entry[x];

    }
```

3.3 Speculation within Instructions

Similar to conditional branches, certain complex instructions may have speculation happen in the internal flow of such instructions which are not branch instructions. One such case is REP string instructions which may speculatively access memory locations that they do not architecturally access.

REP string instructions such as REPE CMPS iterate over memory and terminate when either the size specified is reached or an alternative condition, such as inequality of a data word, is met. Due to speculation, such REP string instruction may transiently execute the string operation beyond the indicated size. While the processor prevents this transient execution from causing architecturally visible effects (such as by restoring the state of architectural registers), those mis-speculated transient string operations can affect the microarchitectural state (for example, by bringing lines into caches).

In this way, a REP CMPS or REP SCAS instruction (for example, REPE CMPS m64, m64) may compare the contents in memory beyond what was specified. Because those instructions terminate their loop based on the data values (for example, non-matching data for REPE CMPS), whether or not the data values beyond the indicated size meet the condition may affect the microarchitectural state (for example, fewer cache lines may be pulled into a cache if the terminating condition is satisfied). An attacker may be able to monitor how many cache lines beyond the indicated size were cached after code in a different security domain executed REP CMPS or REP SCAS. This may allow the attacker to infer whether the values in memory adjacent to the buffers being processed satisfy the condition. This would also be true for a software implementation of `REP CMPS` or `REP SCAS` that did not include bounds check bypass mitigations like `LFENCE` or masking.

Contrary to `REP CMPS` and `REP SCAS`, the execution of `REP MOVS` and `REP STOS` does not include comparison operations and thus does not have this behavior. Intel is not aware of any exploits that result from speculatively executing `REP CMPS` and `REP SCAS` beyond the indicated size.

4.0 Data Speculation

4.1 Overview of Data Speculation

Intel processors implement performance features that allow instructions that depend on the behavior of older instructions to speculatively execute before these older instructions have executed:

- Memory disambiguation predicts whether the address of a memory load overlaps with the yet-unknown address of a preceding memory store to allow speculative execution of the memory load. Misprediction of memory disambiguation can allow for Speculative Store Bypass attacks that transiently access and infer stale data in memory (as described in Section 4.2 Speculative Store Bypass).

- Fast store forwarding predictor allows a memory load to speculatively use the data of a preceding memory store before all store-to-load forwarding conditions are resolved, for example, before a match of the load and store addresses have been resolved.

- The floating-point unit statically predicts floating-point results to be normal to speculatively execute floating-point operations. A microcode assist is triggered to handle denormal/subnormal floating-point results. Floating Point Value Injection is a technique to infer information using the transiently computed floating-point result before a subnormal floating-point microcode assist is triggered and the transient result is cleaned up.

4.2 Speculative Store Bypass

Many Intel processors use memory disambiguation predictors that allow loads to be executed speculatively before it is known whether the load's address overlaps with a preceding store's address. This may happen if a store's address is unknown when the load is ready to execute. If the processor predicts that the load address will not overlap with the unknown store address, the load may execute speculatively. However, if there is indeed an overlap, then the load may consume stale data. When this occurs, the processor will re-execute the load to ensure a correct result.

Through the memory disambiguation predictors, an attacker can cause certain instructions to be executed speculatively and then use the effects for side channel analysis. For example, consider the following scenario:

`K` is a secret asset (for example, a cryptographic key) inside the victim code. The attacker is allowed to know the value of `M`, but not the value of `K`. `X` is a variable in memory. Assuming an attacker can find the following code in a victim application:

1. `X = &K;    // Attacker manages to get variable with address of K stored into pointer X`

   <at some later point>

2. `X = &M;    // Does a store of address of M to pointer X`

3. `Y = Array[*X & 0xFFFF]; // Dereferences address of M which is in pointer X in order to`

```
                        // load from array at index specified by M[15:0]
```

When the above code runs, the load from address `X` that occurs as part of step 3 may execute speculatively and, due to memory disambiguation, initially receive a value of address of `K` instead of the address of `M`. When this value of address of `K` is dereferenced, the array is speculatively accessed with an index of `K[15:0]` instead of `M[15:0]`. The CPU will later re-execute the load from address `X` and use `M[15:0]` as the index into the array. However, the cache movement caused by the earlier speculative access to the array may be analyzed by the attacker to infer information about `K[15:0]`.

As in the previous example, an attacker may be able to discover "confused deputy" code which may allow them to use speculative execution to reveal the value of memory that is not normally accessible to them. In a language-based security environment (for example, a managed runtime), where an attacker is able to influence the generation of code, an attacker may be able to create such a confused deputy. Intel has not currently observed this method in situations where the attacker has to discover such an exploitable confused deputy scenario.

4.3 Speculative Store Bypass Control Mechanisms

Intel has developed mitigation techniques for speculative store bypass. It can be mitigated by software modifications, or if those are not feasible, then the use of Speculative Store Bypass Disable (SSBD), which prevents a load from executing speculatively until the addresses of all older stores are known. Intel recommends using the below mitigations only for managed runtimes or other situations that use language-based security to guard against attacks within an address space.

4.3.1 Software-Based Mitigations

Speculative store bypass can be mitigated through numerous software-based approaches. This section describes two such software-based mitigations: process isolation and the selective use of `LFENCE`.

4.3.1.1 Process Isolation

One approach is to move all secrets into a separate address space from untrusted code. For example, creating separate processes for different websites so that secrets of one website are not mapped into the same address space as code from a different, possibly malicious, website. Similar techniques can be used for other runtime environments that rely on language-based security to run trusted and untrusted code within the same process. This may also be useful as part of a defense-in-depth strategy to prevent trusted code from being manipulated to create a side channel. Protection Keys can also be valuable in providing such isolation. Refer to Section 7.4, Protection Keys, for more information.

4.3.1.2 Using LFENCE to Control Speculative Load Execution

Software can insert an `LFENCE` between a store (for example, the store of address of `M` in step 2 of Section 4.2 Speculative Store Bypass) and the subsequent load (for example, the load that dereferences `X` in step 3 of Section 4.2 Speculative Store Bypass) to prevent the load from executing before the previous store's address is known. The `LFENCE` can also be inserted between the load and any subsequent usage of the data returned which might create a side channel (for example, the access to `Array` in step 3 of Section 4.2 Speculative Store Bypass). Software should not apply this mitigation broadly, but instead should only apply it where there is a realistic risk of an exploit; for example, if an attacker can control the old value in the memory location, there is a realistic chance of the load executing before the store address is known, and there is a disclosure gadget that reveals the contents of sensitive memory.

Other mitigations like inserting register dependencies between a vulnerable load address and the corresponding store address may reduce the likelihood of Speculative Store Bypass Attacks being successful.

4.3.2 Speculative Store Bypass Disable (SSBD)

If the earlier software-based mitigations are not feasible, then employing Speculative Store Bypass Disable (SSBD) will mitigate speculative store bypass.

When SSBD is set, loads will not execute speculatively until the addresses of all older stores are known. This ensures that a load does not speculatively consume stale data values due to bypassing an older store on the same logical processor.

4.3.2.1 Basic Support

Software can disable speculative store bypass on a logical processor by setting IA32_SPEC_CTRL.SSBD to 1.

Both enclave and SMM code will behave as if SSBD is set regardless of the actual value of the MSR bit. The processor will ensure that a load within enclave or SMM code does not speculatively consume stale data values due to bypassing an older store on the same logical processor.

4.3.2.2 Software Usage Guideline

Enabling SSBD can prevent exploits based on speculative store bypass. However, this may reduce performance. Intel provides the following recommendations for the use of such a mitigation.

- Intel recommends software set SSBD for applications and/or execution runtimes relying on language-based security mechanisms. Examples include managed runtimes and just-in-time translators. If software is not relying on language-based security mechanisms, for example because it is using process isolation, then setting SSBD may not be needed.
- Intel is currently not aware of any practical exploit for OSes or other applications that do not rely on language-based security. Intel encourages these users to consider their particular security needs in determining whether to set SSBD outside context of language-based security mechanisms.

These recommendations may be updated in the future.

On Intel® Core™ and Intel® Xeon® processors that enable Intel® Hyper-Threading Technology and do not support enhanced IBRS, setting SSBD on a logical processor may impact the performance of a sibling logical processor on the same core. Intel recommends that the SSBD MSR bit be cleared when in an idle state on such processors.

Operating systems should provide an API through which a process can request it be protected by SSBD mitigation.

VMMs should allow a guest to determine whether to enable SSBD mitigation by providing direct guest access to IA32_SPEC_CTRL.

4.4 Fast Store Forwarding Predictor

Certain Intel processors support a performance feature called Fast Store Forwarding Predictor (FSFP). Based on observing previous behavior, FSFP enables the processor to predict that a store will forward data to a younger load and optimize that case. This optimization may allow the load to speculatively execute with data from an older store before all forwarding conditions (like store-load address match) have been resolved. If data is incorrectly forwarded to the load, the processor will prevent the load from committing to the architectural state and will re-execute the load with the correct data.

4.4.1 Potential for Transient Execution Disclosure Gadgets

As an effective performance optimization, FSFP predicts store-to-load forwarding with high accuracy. These predictions are based on the observed behavior of previous stores and loads. However, misprediction of store-to-load forwarding can happen for specific reasons. For example, either dynamically varying behavior of specific store and load instruction instances, or predictor aliasing between different instruction instances could cause FSFP to mispredict the correct store-to-load forwarding.

As with other forms of speculation, the transient execution of FSFP store-to-load forwarding has the potential to reveal data through a covert channel that would not otherwise be revealed. If a malicious actor is able to identify a disclosure gadget in vulnerable victim code and also induce FSFP store-forwarding speculation as required for the gadget, it may be possible to disclose targeted data accessible to the victim with a “confused-deputy” form of attack.

4.5 Fast Store Forwarding Predictor Controls

Processors that support FSFP maintain several properties to help mitigate the risk of potential transient execution attacks. These properties are intended to enable prediction domain isolation for FSFP, as well as provide mechanisms to disable FSFP where desired.

4.5.1 Cross-Domain and Cross-Thread Training Isolation

The FSFP domain isolation architecture is analogous to the indirect branch prediction domain isolation properties documented for the Indirect Branch Restricted Speculation (IBRS) and Single Thread Indirect Branch Predictors (STIBP) mechanisms.

For processors supporting FSFP, the following domain isolation properties are maintained (regardless of IBRS or STIBP enabling):

- Activity in user mode does not control FSFP prediction in supervisor mode.
- Activity in Virtual Machine Extensions (VMX) guest (non-root) mode does not control FSFP prediction in host (root) mode.

- Activity on one logical processor does not control FSFP prediction on another logical processor.

In addition, the Indirect Branch Predictor Barrier (IBPB) described in section 3.1.2.4 additionally serves as an FSFP prediction barrier. Activity before the barrier on a logical processor does not control FSFP prediction after the barrier.

Note that FSFP does not affect system management mode (SMM) and Intel® Software Guard Extensions (Intel® SGX) enclaves. This is described in more detail below.

4.5.2 Predictive Forwarding Barriers

Processors supporting FSFP prevent predictive forwarding of data values of older stores to younger loads across certain architectural boundaries. If any of the following occur between an older store and a younger load, FSFP will not predictively forward data from that store to that load:

- `LFENCE`
- Change to memory protection key restriction registers `PKRU` or `PKRS`
- Serializing instructions or events

Also note that no store in one privileged predictor mode can be predictively forwarded to a load from another predictor mode. This includes not only mode transitions (such as `SYSCALL`), but also implicit supervisor loads (for example, loads from the global descriptor table (GDT) performed by a segment load instruction).

4.5.3 FSFP Disabled Modes

When FSFP is disabled, the processor does not speculatively execute loads with a value forwarded from a store when the store and load do not have matching addresses. This FSFP-disabled behavior does not intrinsically preclude speculative store bypass.

Both Speculative Store Bypass (SSB) and FSFP are disabled whenever the IA32_SPEC_CTRL.SSBD (see Section 4.3.2 Speculative Store Bypass Disable (SSBD)) MSR bit is set to 1, or inside Intel® SGX enclaves or system management mode (SMM)). All processors that support FSFP also support SSBD.

In addition to SSBD, a finer-grained control is provided for disabling FSFP without disabling SSB. If enumerated by CPUID.(EAX=7,ECX=2).EDX[0], the processor supports setting IA32_SPEC_CTRL.PSFD (bit 7). FSFP will be disabled if either SSBD or PSFD are enabled or if in SMM or Intel SGX mode.

The PSFD bit offset is chosen to align with the Predictive Store Forwarding Disable (PSFD) bit defined here by AMD*.

Going forward, PSFD is expected to be supported on all processors that support FSFP. Some processors may require a microcode update to enable PSFD support. Refer to the Fast Store Forwarding Predictor document for more details.

4.5.4 Software Guidance

In many cases, software environments employing mitigations for Branch Target Injection (Spectre variant 2) and Speculative Store Bypass (Spectre variant 4) may require no further changes for FSFP.

The combination of processor domain isolation, along with software invocation of IBPB on security context switches, can help mitigate the risk of cross-domain FSFP training attacks.

Same-domain exposure for FSFP may be similar to that of Speculative Store Bypass. As with Speculative Store Bypass, language-based security environments (for example, a managed runtime) present the most likely environment where a potential malicious actor may seek to influence the generation of code. Mitigation for such environments are similar to those described for Speculative Store Bypass. Some options include:

- Process isolation
- SSBD
- Targeted mitigation of specific disclosure gadgets (for example, placing `LFENCE` between potential store/load pairs)

Using PSFD to disable FSFP may be of interest in specific environments that are concerned with same-domain FSFP attacks, but which are not concerned about Speculative Store Bypass attacks and where the performance impact of SSBD is a concern.

5.0 Data-Dependent Prefetchers

Besides control and data speculation, Intel processors implement prefetchers that prefetch cache lines from memory based on data values previously loaded or prefetched from memory, for example, data-dependent prefetchers (DDP). While such prefetchers do not create speculative execution paths, they may yet allow an attacker to infer information about loaded data values via cache-based side channels.

Intel processors automatically enforce properties for these prefetchers to mitigate potential security concerns, as well as exposing a disable control, as described in the provided DDP documentation.

6.0 Additional Software Guidance

Due to Speculative Behavior of SWAPGS and Segment Registers, operating systems that use `SWAPGS` on kernel entry may wish to insert an `LFENCE` or serializing instruction after any possible usage of `SWAPGS` instruction and before any instruction which uses the GS register, to prevent speculative execution with the incorrect GS register.

7.0 Related Intel Security Features and Technologies

There are security features and technologies, either present in existing Intel products or planned for future products, which reduce the effectiveness of the attacks mentioned in the previous sections.

7.1 Intel® OS Guard

When Intel® OS Guard, also known as Supervisor-Mode Execution Prevention (SMEP), is enabled, the operating system will not be allowed to directly execute application code, even speculatively. This makes branch target injection attacks on the OS substantially more difficult by forcing the attacker to find gadgets within the OS code. It is also more difficult for an application to train OS code to jump to an OS gadget. All major operating systems enable SMEP support by default.

7.2 Execute Disable Bit

The Execute Disable Bit is a hardware-based security feature that can help reduce system exposure to viruses and malicious code. Execute Disable Bit allows the processor to classify areas in memory where application code can or cannot execute, even speculatively. This reduces the gadget space, increasing the difficulty of branch target injection attacks. All major operating systems enable Execute Disable Bit support by default. Applications are encouraged to only mark code pages as executable.

7.3 Intel® Control flow Enforcement Technology (Intel® CET)

Intel® Control-Flow Enforcement Technology (Intel® CET) is a feature on recent Intel products to protect control-flow integrity against Return-Oriented Programming (ROP) / Call-Oriented Programming (COP) / Jump-Oriented Programming (JOP) style attacks. It provides two main capabilities:

- Shadow stack: A shadow stack is a second independent stack which is used exclusively for control transfer operations. When shadow stacks are enabled, `RET` instructions require that return addresses on the data stack match the address on the shadow stack, which can be used to mitigate ROP attacks.
- Indirect branch tracking (IBT): When IBT is enabled, the processor requires that the instruction at the target of indirect `JMP` or `CALL` instructions is an `ENDBRANCH`. Software must be compiled to place the `ENDBRANCH` instruction at valid targets.

Intel CET also applies restrictions to transient execution to constrain speculative control flow. These restrictions may be relevant for both control-flow speculation and attacker-controlled jump redirection. More details can be found in volume 1 chapter 18 “Control-flow Enforcement Technology (CET)” of the IA-32 Intel® Architecture Software Developer’s Manual.

7.3.1 CET Shadow Stack Speculation Limitations

When CET Shadow Stack is enabled, the processor will not execute instructions, even speculatively, at the loaded target of the return address of a `RET` instruction if that target differs from the predicted target (such as the target predicted by the Return Stack Buffer), and:

- The `RET` address values on the data stack and shadow stack do not match; or
- Those address values may be transient (for example, the values may have been modified by an older speculative store).

7.3.2 Intel CET Indirect Branch Tracking (CET IBT) Speculation Limitations

When CET IBT is enabled, instruction execution will be limited or blocked, even speculatively, if the next instruction is not an `ENDBRANCH` after an indirect `JMP` or `CALL` which sets the IBT tracker state to `WAIT_FOR_ENDBRANCH`. The Tiger Lake implementation of CET limits speculative execution to a small number of instructions (less than eight, with no more than five loads) after a missing `ENDBRANCH`. On Alder Lake, Sapphire Rapids, Raptor Lake, and some future processors, the potential speculation window at a target that does not start with `ENDBRANCH` is limited to two instructions (and typically fewer) with no more than one load.

The intended long-term direction, and behavior on some current implementations (including E-core only products like Alder Lake-N and Arizona Beach), is to completely block the speculative execution of instructions after a missing `ENDBRANCH`.

7.4 Protection Keys

On Intel processors that have both hardware support for mitigating Rogue Data Cache Load (`IA32_ARCH_CAPABILITIES[RDCL_NO]`) and protection keys support (CPUID.7.0.ECX[3]), protection keys can limit the data accessible to a piece of software. This can be used to limit the memory addresses that could be revealed by a branch target injection or bound check bypass attack.

7.5 Supervisor-Mode Execution Prevention (SMEP)

When Intel® OS Guard, also known as Supervisor-Mode Execution Prevention (SMEP), is enabled, the operating system will not be allowed to directly execute user-mode code, even speculatively. This makes branch target injection attacks on the OS substantially more difficult by forcing the attacker to find gadgets within the OS code. All major operating systems enable SMEP support by default.

7.6 Supervisor-Mode Access Prevention (SMAP)

SMAP can be used to limit which memory addresses can be used for a cache-based side channel, by blocking allocation of an application line. This may make it more difficult for an application to perform the attack on the kernel, as it is more challenging for an application to determine whether a kernel line is cached than an application line. On Intel processors that have both hardware support for mitigating Rogue Data Cache Load (`IA32_ARCH_CAPABILITIES[RDCL_NO]`) and SMAP support, loads that cause a page fault due to SMAP will not speculatively return the loaded data even on a L1D cache hit or fill/evict any caches for that address. On processors that have SMAP support but do not enumerate `RDCL_NO`, loads that cause a page fault due to SMAP may speculatively return the loaded data on L1D cache hits but will not fill/evict any caches for that address.

SMAP can also make it more difficult for a malicious user process to exploit disclosure gadgets in the kernel. When SMAP is inactive, transient kernel-mode operations may access maliciously crafted data in the user process, and then subsequent transient kernel-mode operations may unwittingly use this crafted user data to access and expose kernel data through a microarchitectural covert channel. When

SMAP is active, the processor prevents kernel-mode operations from accessing user-mode data, and therefore also prevents user-mode data from participating in a kernel-mode disclosure gadget.

7.7 Linear Address Space Separation

Linear Address Space Separation (LASS), described in chapter 9 of the Intel® Architecture Instruction Set Extensions and Future Features programming reference, prevents user mode code from causing page walks and Translation Lookaside Buffer (TLB) fills for supervisor addresses, and (when SMAP is enabled and effective) also provides similar limitations for supervisor code attempting to access user mode addresses.

When LASS is not used, such page walks and TLB fills may allow a user mode attacker to infer which linear addresses in supervisor space are mapped; which may lead to breaking Kernel Address Space Layout Randomization (KASLR).

LASS can also help reduce the risk of speculative execution associated with other new features. For example, when Linear Address Masking (LAM) (described in chapter 6 of the Intel® Architecture Instruction Set Extensions and Future Features programming reference) is enabled, addresses that would otherwise be non-canonical may be valid pointers after masking is applied, providing an address-translation covert channel for a wider range of values. Although this is not a vulnerability, it could potentially be used as part of other attacks. When LAM is enabled for a user application, enabling LASS and SMAP in supervisor code restricts the potential use of this covert channel, acting as a defense-in-depth mitigation. Intel recommends operating system software enable LASS when possible.

7.8 Intel® Advanced Performance Extensions (Intel® APX)

Intel® Advanced Performance Extensions (Intel® APX) introduces a family of instructions called `CFCMOVcc` that do not trigger a fault even when the memory operand is invalid, or when page permission checks fail.

`CMOVcc` (whether with `REX2`, `EVEX` prefix or no prefix) do not suppress memory faults or zero the destination register when the condition is false. This differs from `CFCMOVcc` which will suppress memory faults and, when used without `NDD`, zero the destination register when the condition is false.

Additionally, the `CFCMOVcc`, `REX2`/`EVEX CMOVcc`, `CCMP` and `CTEST` instructions, like non-APX `CMOVcc` and `SETcc`, will not predict whether the condition is true or false or predict source flags but will instead use the direction specified by their source flags, even speculatively. Any new feature that changes this (such as by adding conditional direction prediction) would require software to explicitly enable it (for example, by using new instructions).

When `CFCMOVcc` is executed in an environment other than an Intel SGX enclave and the memory access faults, a value of zero may be written transiently to the destination register. When `CFCMOVcc` is executed in an Intel SGX enclave and the memory access faults, data is not written transiently to the destination register. This behavior is intended to prevent vulnerabilities similar to *LVI zero data*, in which an Intel SGX enclave could invertedly load malicious data from linear address zero during transient execution. Such behavior may present a concern for Intel SGX enclaves since the Intel SGX threat model (unlike in other environments) may include a privileged attacker who is able to control the mapping of linear address zero. For more information, see Load Value Injection.

Do note that almost all instructions, including those discussed here, may be affected by control flow speculation (for example, conditional jumps) or data speculation that affect their sources. These new instructions (`CFCMOVcc`, `REX2`/`EVEX CMOVcc`, `CCMP` and `CTEST`) are expected to be as effective for mitigating bound check bypass or avoiding timing side channels in constant time code as existing instructions used for these purposes – such as non-APX forms of `CMOVcc`/`SETcc`, `AND`, `ADC`, or `SBB`.

8.0 CPUID Enumeration and Architectural MSRs

CPUID Enumeration and Architectural MSRs describes processor support for mitigation mechanisms as enumerated using the CPUID instruction and several architectural MSRs.

9.0 Resources

- Speculative Execution Side Channel Mitigations
- Intel Analysis of Speculative Execution Side Channels
- CPUID Enumeration and Architectural MSRs
- Refined Speculation Execution Terminology
- Retpoline: A Branch Target Injection Mitigation
- Analyzing Potential Bounds Check Bypass Vulnerabilities
- Host Firmware Speculative Execution Side Channel Mitigations
- Fast Store Forwarding Predictor
- Software Security Guidance
- Intel Security Center
- Configuring Spectre mitigations in the Linux kernel

10.0 Appendix: Summary Table of How Mitigation Techniques are Used

Vulnerability	Intel Naming	Mitigation Techniques
Spectre v2	Branch Target Injection	IBRS, eIBRS, Retpoline
Retbleed	Return Stack Buffer Underflow	Retpoline+ call Depth Tracking, IBRS
Native BHI	Branch History Injection	BHI_DIS_S, BHB clearing sequence for branch history clearing
Training Solo	Indirect Target Selection	Relocating vulnerable indirect branches to safe thunk
VMSCAPE	VMSCAPE	IBPB, user space IBRS, BHB clearing sequence for branch history clearing