# CWE Detail – CWE-1423

## Description

Shared microarchitectural predictor state may allow code to influence  
 transient execution across a hardware boundary, potentially exposing  
 data that is accessible beyond the boundary over a covert channel.

## Extended Description

Many commodity processors have Instruction Set Architecture (ISA)  
 features that protect software components from one another. These  
 features can include memory segmentation, virtual memory, privilege  
 rings, trusted execution environments, and virtual machines, among  
 others. For example, virtual memory provides each process with its own  
 address space, which prevents processes from accessing each other's  
 private data. Many of these features can be used to form  
 hardware-enforced security boundaries between software components. When separate software components (for example, two processes) share  
 microarchitectural predictor state across a hardware boundary, code in  
 one component may be able to influence microarchitectural predictor  
 behavior in another component. If the predictor can cause transient  
 execution, the shared predictor state may allow an attacker to  
 influence transient execution in a victim, and in a manner that could  
 allow the attacker to infer private data from the victim by monitoring  
 observable discrepancies (CWE-203) in a covert channel [REF-1400]. Predictor state may be shared when the processor transitions from one  
 component to another (for example, when a process makes a system call  
 to enter the kernel). Many commodity processors have features which  
 prevent microarchitectural predictions that occur before a boundary  
 from influencing predictions that occur after the boundary. Predictor state may also be shared between hardware threads, for  
 example, sibling hardware threads on a processor that supports  
 simultaneous multithreading (SMT). This sharing may be benign if the  
 hardware threads are simultaneously executing in the same software  
 component, or it could expose a weakness if one sibling is a malicious  
 software component, and the other sibling is a victim software  
 component. Processors that share microarchitectural predictors between  
 hardware threads may have features which prevent microarchitectural  
 predictions that occur on one hardware thread from influencing  
 predictions that occur on another hardware thread. Features that restrict predictor state sharing across transitions or  
 between hardware threads may be always-on, on by default, or may  
 require opt-in from software.

## Threat-Mapped Scoring

Score: 1.8

Priority: P4 - Informational (Low)

## Observed Examples (CVEs)

**•** CVE-2017-5754: (Branch Target Injection, BTI, Spectre v2). Shared  
 microarchitectural indirect branch predictor state may allow code to  
 influence transient execution across a process, VM, or privilege  
 boundary, potentially exposing data that is accessible beyond the  
 boundary.

**•** CVE-2022-0001: (Branch History Injection, BHI, Spectre-BHB). Shared  
 branch history state may allow user-mode code to influence transient  
 execution in the kernel, potentially exposing kernel data over a  
 covert channel.

**•** CVE-2021-33149: (RSB underflow, Retbleed). Shared return stack buffer  
 state may allow code that executes before a prediction barrier to  
 influence transient execution after the prediction barrier,  
 potentially exposing data that is accessible beyond the barrier over a  
 covert channel.

## Modes of Introduction

**•** Architecture and Design: This weakness can be introduced during hardware architecture and  
 design if predictor state is not properly isolated between modes (for  
 example, user mode and kernel mode), if predictor state is not  
 isolated between hardware threads, or if it is not isolated between  
 other kinds of execution contexts supported by the processor.

**•** Implementation: This weakness can be introduced during system software  
 implementation if predictor-state-sanitizing operations (for example,  
 the indirect branch prediction barrier on Intel x86) are not invoked  
 when switching from one context to another.

**•** System Configuration: This weakness can be introduced if the system has not been  
 configured according to the hardware vendor's recommendations for  
 mitigating the weakness.

## Common Consequences

**•** Impact: Read Memory — Notes:

## Potential Mitigations

**•** Architecture and Design: The hardware designer can attempt to prevent transient  
 execution from causing observable discrepancies in specific covert  
 channels. (Effectiveness: N/A)

**•** Architecture and Design: Hardware designers may choose to use microarchitectural  
 bits to tag predictor entries. For example, each predictor entry may  
 be tagged with a kernel-mode bit which, when set, indicates that the  
 predictor entry was created in kernel mode. The processor can use this  
 bit to enforce that predictions in the current mode must have been  
 trained in the current mode. This can prevent malicious cross-mode  
 training, such as when user-mode software attempts to create predictor  
 entries that influence transient execution in the kernel. Predictor  
 entry tags can also be used to associate each predictor entry with the  
 SMT thread that created it, and thus the processor can enforce that  
 each predictor entry can only be used by the SMT thread that created  
 it. This can prevent an SMT thread from using predictor entries  
 crafted by a malicious sibling SMT thread. (Effectiveness: Moderate)

**•** Architecture and Design: Hardware designers may choose to sanitize  
 microarchitectural predictor state (for example, branch prediction  
 history) when the processor transitions to a different context, for  
 example, whenever a system call is invoked. Alternatively, the  
 hardware may expose instruction(s) that allow software to sanitize  
 predictor state according to the user's threat model. For example,  
 this can allow operating system software to sanitize predictor state  
 when performing a context switch from one process to another. (Effectiveness: Moderate)

**•** Implementation: System software can mitigate this weakness by invoking  
 predictor-state-sanitizing operations (for example, the indirect  
 branch prediction barrier on Intel x86) when switching from one  
 context to another, according to the hardware vendor's  
 recommendations. (Effectiveness: Moderate)

**•** Build and Compilation: If the weakness is exposed by a single instruction (or a  
 small set of instructions), then the compiler (or JIT, etc.) can be  
 configured to prevent the affected instruction(s) from being  
 generated. One prominent example of this mitigation is retpoline  
 ([REF-1414]). (Effectiveness: Limited)

**•** Build and Compilation: Use control-flow integrity (CFI) techniques to constrain  
 the behavior of instructions that redirect the instruction pointer,  
 such as indirect branch instructions. (Effectiveness: Moderate)

**•** Build and Compilation: Use software techniques (including the use of  
 serialization instructions) that are intended to reduce the number of  
 instructions that can be executed transiently after a processor event  
 or misprediction. (Effectiveness: Incidental)

**•** System Configuration: Some systems may allow the user to disable predictor  
 sharing. For example, this could be a BIOS configuration, or a  
 model-specific register (MSR) that can be configured by the operating  
 system or virtual machine monitor. (Effectiveness: Moderate)

**•** Patching and Maintenance: The hardware vendor may provide a patch to, for example,  
 sanitize predictor state when the processor transitions to a different  
 context, or to prevent predictor entries from being shared across SMT  
 threads. A patch may also introduce new ISA that allows software to  
 toggle a mitigation. (Effectiveness: Moderate)

**•** Documentation: If a hardware feature can allow microarchitectural  
 predictor state to be shared between contexts, SMT threads, or other  
 architecturally defined boundaries, the hardware designer may opt to  
 disclose this behavior in architecture documentation. This  
 documentation can inform users about potential consequences and  
 effective mitigations. (Effectiveness: High)

**•** Requirements: Processor designers, system software vendors, or other  
 agents may choose to restrict the ability of unprivileged software to  
 access to high-resolution timers that are commonly used to monitor  
 covert channels. (Effectiveness: N/A)

## Applicable Platforms

**•** None (Class: Not Language-Specific, Prevalence: Undetermined)

## Demonstrative Examples

**•** To successfully exploit this code sequence to disclose the victim's  
 private data, the attacker must also be able to find an indirect  
 branch site within the victim, where the attacker controls the values  
 in edi and ebx, and the attacker knows the value in edx as shown above  
 at the indirect branch site. A proof-of-concept cross-thread BTI attack might proceed as follows: The attacker thread and victim thread must be co-scheduled on the same physical processor core. The attacker thread must train the shared branch predictor so that  
 when the victim thread reaches indirect\_branch\_site, the jmp  
 instruction will be predicted to target example\_code\_sequence instead  
 of the correct architectural target. The training procedure may vary  
 by processor, and the attacker may need to reverse-engineer the branch  
 predictor to identify a suitable training algorithm. This step assumes that the attacker can control some values in the  
 victim program, specifically the values in edi and ebx at  
 indirect\_branch\_site. When the victim reaches indirect\_branch\_site the  
 processor will (mis)predict example\_code\_sequence as the target and  
 (transiently) execute the adc instructions. If the attacker chooses  
 ebx so that `ebx = m 0x13BE13BD - edx, then the first adc will load 32 bits from  
 address m in the victim's address space and add \*m (the data loaded from)  
 to the attacker-controlled base address in edi. The second  
 adc instruction accesses a location in memory whose address corresponds  
 to \*m`. The adversary uses a covert channel analysis technique such as  
 Flush+Reload ([REF-1416]) to infer the value of the victim's private data  
 \*m.

**•** N/A