# Microarchitectural Data Sampling (MDS) mitigation

## Overview

Microarchitectural Data Sampling (MDS) is a family of side channel attacks on internal buffers in Intel CPUs. The variants are:

- Microarchitectural Store Buffer Data Sampling (MSBDS) (CVE-2018-12126)
- Microarchitectural Fill Buffer Data Sampling (MFBDS) (CVE-2018-12130)
- Microarchitectural Load Port Data Sampling (MLPDS) (CVE-2018-12127)
- Microarchitectural Data Sampling Uncacheable Memory (MDSUM) (CVE-2019-11091)

MSBDS leaks Store Buffer Entries which can be speculatively forwarded to a dependent load (store-to-load forwarding) as an optimization. The forward can also happen to a faulting or assisting load operation for a different memory address, which can be exploited under certain conditions. Store buffers are partitioned between Hyper-Threads so cross thread forwarding is not possible. But if a thread enters or exits a sleep state the store buffer is repartitioned which can expose data from one thread to the other.

MFBDS leaks Fill Buffer Entries. Fill buffers are used internally to manage L1 miss situations and to hold data which is returned or sent in response to a memory or I/O operation. Fill buffers can forward data to a load operation and also write data to the cache. When the fill buffer is deallocated it can retain the stale data of the preceding operations which can then be forwarded to a faulting or assisting load operation, which can be exploited under certain conditions. Fill buffers are shared between Hyper-Threads so cross thread leakage is possible.

MLPDS leaks Load Port Data. Load ports are used to perform load operations from memory or I/O. The received data is then forwarded to the register file or a subsequent operation. In some implementations the Load Port can contain stale data from a previous operation which can be forwarded to faulting or assisting loads under certain conditions, which again can be exploited eventually. Load ports are shared between Hyper-Threads so cross thread leakage is possible.

MDSUM is a special case of MSBDS, MFBDS and MLPDS. An uncacheable load from memory that takes a fault or assist can leave data in a microarchitectural structure that may later be observed using one of the same methods used by MSBDS, MFBDS or MLPDS.

## Exposure assumptions

It is assumed that attack code resides in user space or in a guest with one exception. The rationale behind this assumption is that the code construct needed for exploiting MDS requires:

- to control the load to trigger a fault or assist
- to have a disclosure gadget which exposes the speculatively accessed data for consumption through a side channel.
- to control the pointer through which the disclosure gadget exposes the data

The existence of such a construct in the kernel cannot be excluded with 100% certainty, but the complexity involved makes it extremly unlikely.

There is one exception, which is untrusted BPF. The functionality of untrusted BPF is limited, but it needs to be thoroughly investigated whether it can be used to create such a construct.

## Mitigation strategy

All variants have the same mitigation strategy at least for the single CPU thread case (SMT off): Force the CPU to clear the affected buffers.

This is achieved by using the otherwise unused and obsolete VERW instruction in combination with a microcode update. The microcode clears the affected CPU buffers when the VERW instruction is executed.

For virtualization there are two ways to achieve CPU buffer clearing. Either the modified VERW instruction or via the L1D Flush command. The latter is issued when L1TF mitigation is enabled so the extra VERW can be avoided. If the CPU is not affected by L1TF then VERW needs to be issued.

If the VERW instruction with the supplied segment selector argument is executed on a CPU without the microcode update there is no side effect other than a small number of pointlessly wasted CPU cycles.

This does not protect against cross Hyper-Thread attacks except for MSBDS which is only exploitable cross Hyper-thread when one of the Hyper-Threads enters a C-state.

The kernel provides a function to invoke the buffer clearing:

```
mds_clear_cpu_buffers()
```

The mitigation is invoked on kernel/userspace, hypervisor/guest and C-state (idle) transitions.

As a special quirk to address virtualization scenarios where the host has the microcode updated, but the hypervisor does not (yet) expose the MD_CLEAR CPUID bit to guests, the kernel issues the VERW instruction in the hope that it might actually clear the buffers. The state is reflected accordingly.

According to current knowledge additional mitigations inside the kernel itself are not required because the necessary gadgets to expose the leaked data cannot be controlled in a way which allows exploitation from malicious user space or VM guests.

## Kernel internal mitigation modes

| | |
|---|---|
| off | Mitigation is disabled. Either the CPU is not affected or mds=off is supplied on the kernel command line |
| full | Mitigation is enabled. CPU is affected and MD_CLEAR is advertised in CPUID. |
| vmwerv | Mitigation is enabled. CPU is affected and MD_CLEAR is not advertised in CPUID. That is mainly for virtualization scenarios where the host has the updated microcode but the hypervisor does not expose MD_CLEAR in CPUID. It's a best effort approach without guarantee. |

If the CPU is affected and mds=off is not supplied on the kernel command line then the kernel selects the appropriate mitigation mode depending on the availability of the MD_CLEAR CPUID bit.

## Mitigation points

### 1. Return to user space

When transitioning from kernel to user space the CPU buffers are flushed on affected CPUs when the mitigation is not disabled on the kernel command line. The migitation is enabled through the static key mds_user_clear.

The mitigation is invoked in prepare_exit_to_usermode() which covers all but one of the kernel to user space transitions. The exception is when we return from a Non Maskable Interrupt (NMI), which is handled directly in do_nmi().

(The reason that NMI is special is that prepare_exit_to_usermode() can
enable IRQs. In NMI context, NMIs are blocked, and we don't want to enable IRQs with NMIs blocked.)

### 2. C-State transition

When a CPU goes idle and enters a C-State the CPU buffers need to be cleared on affected CPUs when SMT is active. This addresses the repartitioning of the store buffer when one of the Hyper-Threads enters a C-State.

When SMT is inactive, i.e. either the CPU does not support it or all sibling threads are offline CPU buffer clearing is not required.

The idle clearing is enabled on CPUs which are only affected by MSBDS and not by any other MDS variant. The other MDS variants cannot be protected against cross Hyper-Thread attacks because the Fill Buffer and the Load Ports are shared. So on CPUs affected by other variants, the idle clearing would be a window dressing exercise and is therefore not activated.

The invocation is controlled by the static key mds_idle_clear which is switched depending on the chosen mitigation mode and the SMT state of the system.

The buffer clear is only invoked before entering the C-State to prevent that stale data from the idling CPU from spilling to the Hyper-Thread sibling after the store buffer got repartitioned and all entries are available to the non idle sibling.

When coming out of idle the store buffer is partitioned again so each sibling has half of it available. The back from idle CPU could be then speculatively exposed to contents of the sibling. The buffers are flushed either on exit to user space or on VMENTER so malicious code in user space or the guest cannot speculatively access them.

The mitigation is hooked into all variants of halt()/mwait(), but does not cover the legacy ACPI IO-Port mechanism because the ACPI idle driver has been superseded by the intel_idle driver around 2010 and is preferred on all affected CPUs which are expected to gain the MD_CLEAR functionality in microcode. Aside of that the IO-Port mechanism is a legacy interface which is only used on older systems which are either not affected or do not receive microcode updates anymore.