WHITE PAPER

Firmware Security

DMA Protection in UEFI

A Tour Beyond BIOS: Using IOMMU for DMA Protection in UEFI Firmware

This paper presents the idea of using an input–output memory management unit (IOMMU) to resist Direct Memory Access (DMA) attacks in firmware. The example presented uses Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d), and the concept can be applied to other IOMMU engines.

Authors:

Jiewen Yao

Intel Corporation

Vincent J. Zimmer

Intel Corporation

Star Zeng

Intel Corporation

Table of Contents

Overview

Prerequisites

This paper assumes that the audience has firmware development experience with the Unified Extensible Firmware Interface (UEFI) and the TianoCore EFI Development Kit (EDK II). This paper also assumes the audience has basic knowledge of DMA and the Peripheral Component Interconnect (PCI) specification.

Introduction to DMA and IOMMU

The need for Direct Memory Access (DMA) has been clearly described before.

> *On the higher-bandwidth buses, a great deal of information is flowing through the channel every second. Normally, the processor is required to control the transfer of this information. In essence, the processor is a "middleman," and as with many similar cases in the real world, it is far more efficient to "cut out" the middleman and perform the transfer directly. This is done by having capable devices take control of the bus and do the work themselves; devices that can do this are called bus masters. ...Bus mastering is also called "first party" DMA since the work is controlled by the device doing the transfer...*
>
> *Bus mastering is the capability of devices on the PCI bus (other than the system chipset, of course) to take control of the bus and perform transfers directly. ...PCI supports full device bus mastering, and provides bus arbitration facilities through the system chipset.*

[BusMastering]

However, bus mastering also opens a door for the system to be attacked. Much research [DMA1] [DMA2] [DMA3] [DMA4] [DMA5] [DMA6] [DMA7] already showed how to use DMA to attack a system. By plugging a malicious device, a hacker may read or write the system memory to break confidentiality or integrity. A software check is not useful because the memory access transaction is initiated by a device not a CPU.

Some OS vendors have expressed a desire to mitigate such attacks through blocking drivers [WindowsDMA], testing [HSTI] and firmware requirements [WindowsRequirement] – System.Fundamentals.Firmware.NoExternalDMAOnBoot. "All external DMA ports must be off by default until the OS explicitly powers them through related controller(s)."

A platform may have many ways to block DMA, such as PCI Bus Master Enable (BME) bit [PCI][PCIExpress], DMA Protected Region (DPR) [TXT], or Protected Memory Region (PMR) [Intel VT-d]. However, each solution has some limitations. The BME is an all-or-none configuration for a PCI device. The DPR can only be set for a region below SMRAM. The PMR can only define 2 regions – one below 4GiB and the other above 4GiB – for all devices. They do not provide fine granularity for configuration.

Intel Virtualization Technology for Direct I/O [Intel VT-d] is an I/O memory management unit (IOMMU) designed for the VMM (Virtual Machine Monitor), to support I/O virtualization. Since Intel VT-d has the capability of fine-grained access control per device, it is a better mitigation for DMA attacks.

Other system architectures also have similar IOMMU capability, such as [AMD IOMMU], [ARM SMMU].

Introduction to EDK II

The UEFI specification [UEFI] has been adopted in most server, PC, mobile and tablet systems. EDK II is the open source implementation for UEFI. EDK II has the DXE (Driver eXecution Environment) stage to produce the UEFI interface. The DXE core, like an OS kernel, has a dispatcher and scheduler to run other drivers and produces services and functions that are architecturally required by UEFI. In the DXE phase, some drivers are responsible for system integrity. For example, the DXE security architecture protocol will check the signature of UEFI images when UEFI secure boot is enabled. The DXE core and DXE security architecture protocol maintain important data structures and they need to be protected.

Threat Model

From a system perspective, there are two types of DMA threats:

- DMA requests from an external device (external DMA), such as a PCI plug-in card (graphic card, network card, storage card) or external device (Thunderbolt device, 1394 device)
- DMA requests from an internal device (internal DMA), such as the USB device on the board, ACPI device on the board.

In order to mitigate the threat from the unauthorized external DMA, provided that the external DMA device is a PCI device, the platform firmware could disable the PCI BME bit. This solution has below limitations:

1) This requirement precludes the use of the external storage device as the OS boot media. The boot device (no matter an internal device or an external device) requires access to all system memory. There is no fine granularity control of which memory can be accessed.
2) UEFI BIOS supports booting different devices one after another. Depending on the interpretation followed by a platform BDS driver, it may enable DMA for the external device when the system starts booting the option ROM on this device and disable DMA for the device the option ROM returns. This increases the complexity of platform BDS code, because in most implementations, the platform BDS does not "disconnect" devices. In some special case, the platform BDS may "connect" all storage devices (not only boot devices), such

as booting to a UEFI shell, unlocking OPAL devices, and performing TPer Reset.

3) Since an external device may be malicious, disabling bus mastering at the endpoint device itself is not sufficient. This would require the endpoint to correctly honor the BME setting and attempting DMA without an external entity preventing it from doing so. Instead, BME can be disabled for PCI root bridges. In this state, the root bridges would be trusted to not forward any DMA transactions, even if they are made by endpoint devices. Unfortunately, this is even less granular, disabling all DMA to all devices attached to the bridge, including most boot devices. This method can still be useful for establishing DMA protection during sensitive pre-boot processing, however.

In order to mitigate the threat from unauthorized internal DMA as well as DMA from non-PCI devices, platform firmware may configure the IOMMU. In addition to protecting from non-PCI devices that do not have BME, the IOMMU solution provides some other advantages as well:

1) The IOMMU supports fine grained memory access control. Device access can be restricted to only a given DMA buffer, and access requests outside that buffer will fail.
2) The IOMMU supports device-level isolation. A device can only access its own DMA buffer. The device cannot access the DMA buffer for other devices.

Regardless of which mitigations are implemented by system firmware, DMA protections need to be enabled early and be configured to enforce least privilege to devices. If PCI BME is used, bus mastering must be disabled at the root bridge for the entire duration critical early processing until the platform security configuration is locked down. Even then PCI BME should be cleared for external devices, unless needed during boot. If IOMMU is used, the IOMMU must be setup before other devices have the ability to access system memory. This might be very early in the PEI phase.

Goal and Motivation

While many UEFI BIOS implementations report the IOMMU-related ACPI table, most implementations do not actually enable the IOMMU engine to provide DMA protection for firmware during its execution. These implementations are simply reporting the platform capabilities. In this paper, we will describe how to enable the IOMMU in a UEFI BIOS to mitigate DMA attacks against firmware.

This document uses Intel VT-d as the example to explain how to produce an IOMMU protocol.

Sample:
https://github.com/tianocore/edk2/tree/master/IntelSiliconPkg/Feature/VTd/IntelVTdDxe

This concept can be used for the other IOMMU capabilities, such as AMD IOMMU or ARM SMMU. We use the PCI subsystem to explain how to consume the IOMMU protocol, and this concept can be used for other devices, such as the ACPI low power subsystem (LPSS) - universal asynchronous receiver/transmitter (UART), Inter-Integrated Circuit (I2C), (Serial Peripheral Interface) SPI, (Secure Digital Input Output) SDIO etc.

Introduction to Intel® Virtualization Technology

Intel VT and Intel VT-d

Intel® Virtualization Technology (Intel® VT) is a processor feature that enables users to run multiple operating systems and applications in independent partitions. Each partition behaves like a virtual machine (VM) and provides isolation and protection across partitions. A Virtual-Machine Monitor (VMM) acts as a host and has full control of the processor(s) and other platform hardware.

Intel® Virtualization Technology for Directed I/O (Intel® VT-d) provides IO virtualization, which includes the following capabilities:

- I/O device assignment: for flexibly assigning I/O devices to VMs and extending the protection and isolation properties of VMs for I/O operations.

- DMA remapping: for supporting address translations for Direct Memory Accesses (DMA) from devices.
- Interrupt remapping: for supporting isolation and routing of interrupts from devices and external interrupt controllers to appropriate VMs.
- Interrupt posting: for supporting direct delivery of virtual interrupts from devices and external interrupt controllers to virtual processors.
- Reliability: for recording and reporting of DMA and interrupt errors to system software that may otherwise corrupt memory or impact VM isolation.

Intel VT and Intel VT-d are different features. They can be enabled independently from a technical perspective. A solution may activate Intel VT without Intel VT-d, if there is no DMA concern. A solution may only activate Intel VT-d if there is no need for system partitioning.

For example, an OS can use the DMA remapping feature for purposes listed below:

- OS Protection: An OS may define a domain containing its critical code and data structures, and restrict access to this domain from all I/O devices in the system. This allows the OS to limit erroneous or unintended corruption of its data and code through incorrect programming of devices by device drivers, thereby improving OS robustness and reliability.
- Feature Support: An OS may use domains to better manage DMA from legacy devices to high memory (For example, 32-bit PCI devices accessing memory above 4GB). This is achieved by programming the I/O page-tables to remap DMA from these devices to high memory. Without such support, software must resort to data copying through OS "bounce buffers".
- DMA Isolation: An OS may manage I/O by creating multiple domains and assigning one or more I/O devices to each domain. Each device-driver explicitly registers its I/O buffers with the OS, and the OS assigns these I/O buffers to specific domains, using hardware to enforce DMA domain protection.
- Shared Virtual Memory: For devices supporting PCI-Express capabilities – PASID (Process Address Space ID) [PCIExpress], OS may use the DMA remapping hardware capabilities to share virtual address space of application processes with I/O devices. Shared virtual memory along with support for I/O page-faults enable application programs to freely pass arbitrary data-structures to devices such as graphics processors or accelerators, without the overheads of pinning and marshalling of data.

This paper will only focus on the DMA remapping feature. For more information on Intel VT and Intel VT-d, such as interrupt remapping or interrupt posing, please refer to the [Intel VT-d] specification.

DMA Remapping

A key concept for DMA remapping is address translation. Figure 1, taken from the [Intel VT-d] specification, illustrates this concept. The left hand side is for processor virtualization, and the right hand side is for IO virtualization. On the right side, both Device 1 and Device 2 want to access memory address 0x4000. The DMA remapping unit (DMA memory management) can map the guest physical address (GPA) to the host physical address (HPA). As the final result, Device 1 accesses host physical address 0x6000 and Device 2 accesses host physical address 0x3000.

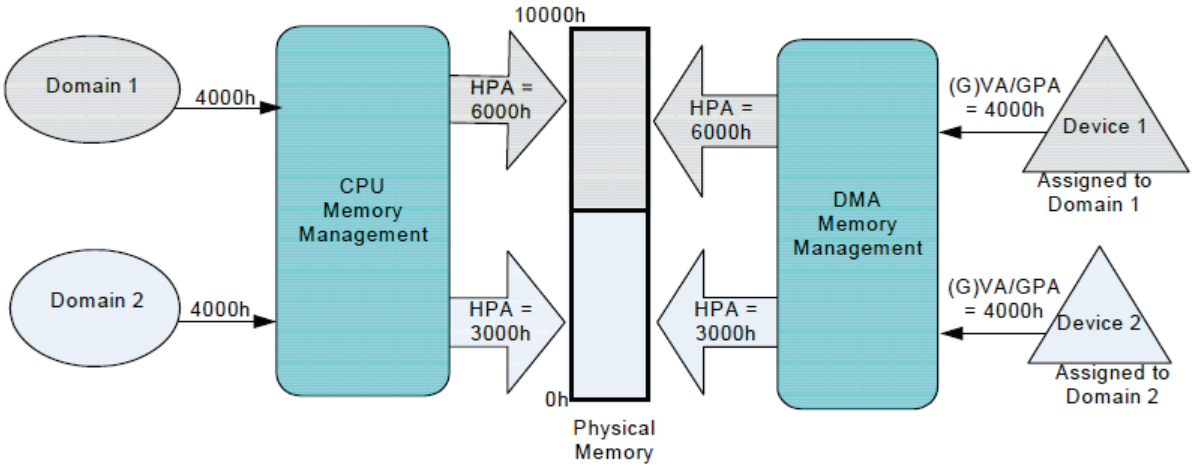

Figure 1 - DMA Address Translation

Address Translation Structures

In order to allow a device to find the proper host physical address, the DMA remapping unit must setup a "translation table" for each device.

Each inbound request appearing at the address-translation hardware is required to identify the device originating the request. The attribute identifying the originator of an I/O transaction is referred to as the "**source-id**". For PCI-Express devices, the source-id is composed of its PCI Bus/Device/Function number.

Firstly, the system has **root-table** functions as the top level structure to map devices to their respective domains. The location of root-table is in an Intel VT-d register - **Root Table Address Register**. The root table contains 256 root-entries to cover the PCI bus number space (0-255). Each **root-entry** contains **context-table** pointer. Each context-table contains 256 entries, with each entry corresponding to a PCI device function on the bus. Each **context-entry** contains a **second level page-table** pointer. As such for each PCI device (with the bus number, device number and function number), there is an address translation structure (second level page-table) associated. See Figure 2 below, taken from the [Intel VT-d] specification.

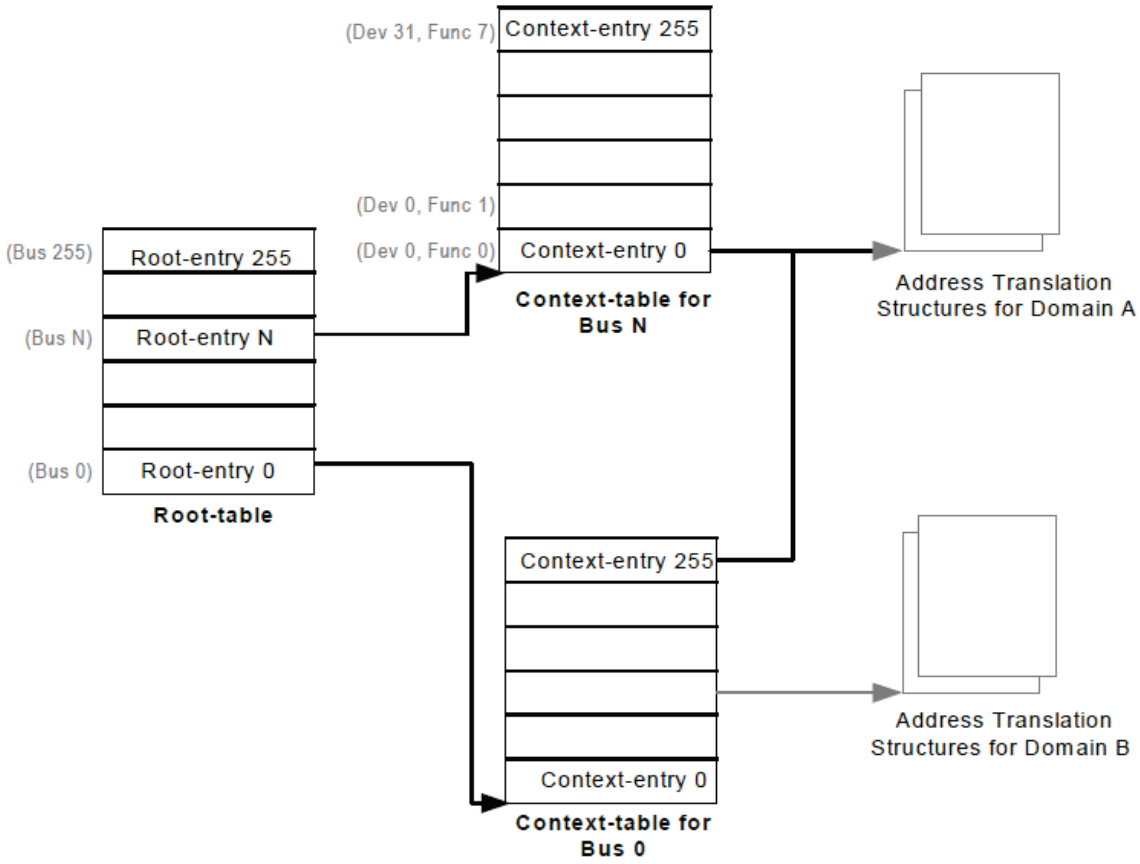

Figure 2 - Device to Domain Mapping Structures using Root-Table

Secondly, the address translation structure for the **second level page table** is similar to the page able for a CPU. The difference is that the second level page table uses X (execution), W (write), R (read) bit, while the CPU page table uses XD (executable), Read/Write(RW), P(Present) bit for the execution, write and read access control. The entry names are changed to **SL-PML4E**, **SL-PDPE**, **SL-PDE**, and **SL-PTE**. See Figure 3 and Figure 4 from the [Intel VT-d] specification.

The full translation process is:

1) Parse the root-table and the context table (Illustrated in Figure 3)
2) Parse the second level page table. (Illustrated in Figure 4)

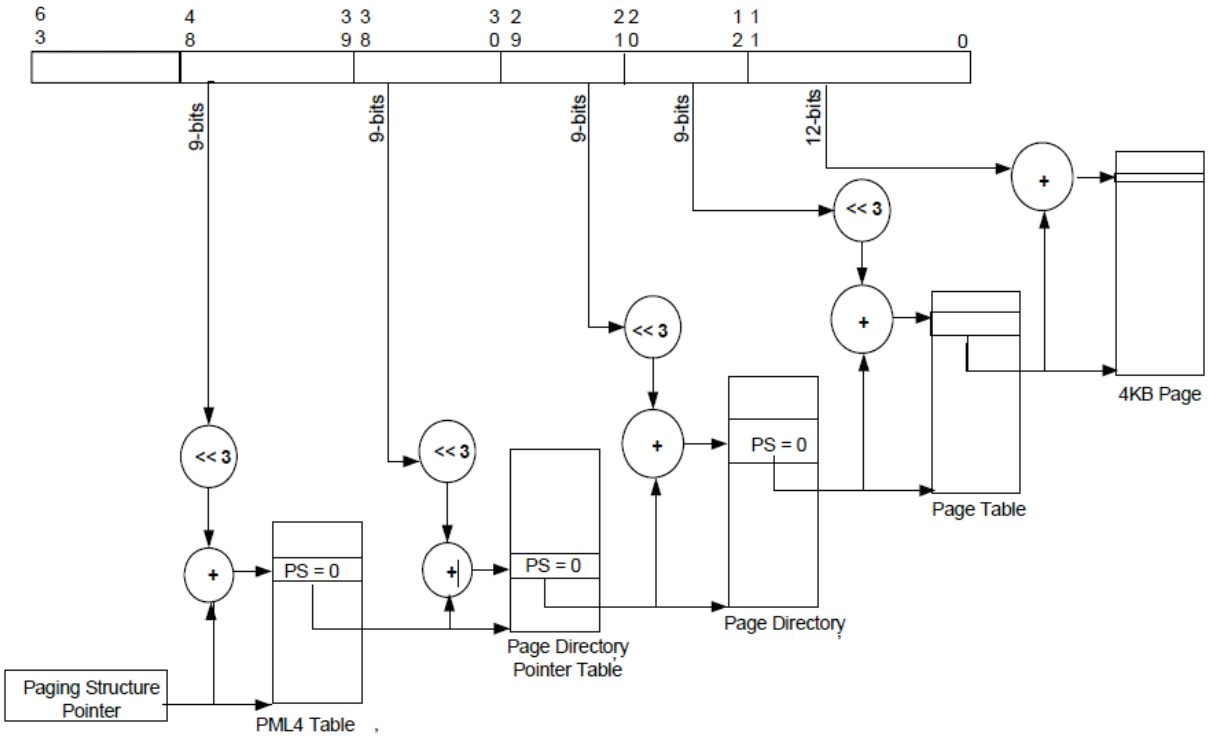

Figure 3 - Address Translation to a 4-Kbyte Page

63	62	61	60	59	58	57	56	55	54	53	52	51	HAW	HAW-1	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Ign	Rsvd	Ignored											Rsvd.	Address of Second-level-page-directory-pointer table																								Rsvd	Ign	Rsvd	Ign				X[1]	W	R	SL-PML4E
Ign	TM	Ignored											Rsvd.	Address of 1GB page frame											Reserved												SNP	Ign	1	IPAT	EMT[2]	X	W	R	SL-PDPE: 1GB page[3]
Ign	Rsvd	Ignored											Rsvd.	Address of second-level-page-directory table																								Rsvd	Ign	0	Ign		X	W	R	SL-PDPE: page directory
Ign	TM	Ignored											Rsvd.	Address of 2MB page frame											Reserved												SNP	Ign.	1	IPAT	EMT	X	W	R	SL-PDE: 2MB page[5]
Ign	Rsvd	Ignored											Rsvd.	Address of second-level-page table																								Rsvd	Ign	0	Ign		X	W	R	SL-PDE: page table
Ign	TM	Ignored											Rsvd.	Address of 4KB page frame																								SNP	Ign		IPAT	EMT	X	W	R	SL-PTE: 4KB page

Figure 4 - Format for Second-Level Paging Entries

If the Intel VT-d Root Table Address Register has an extended bit set, the table is an **extended-root-table**, which points to an **extended-context-table**. Each **extended-context-entry** contains a **PASID-table** (Process Address Space ID) or a **second level page table** (without PASID). The PASID-table has a **PASID-entry**, which points to a **first level page table**. The first level page table uses the

same paging structure as Intel® 64 processors in 64-bit mode.

Devices report the support for requests-with-PASID through the PCI-Express PASID Capability structure. PASID Capability allows the software to query and control if the endpoint can issue requests-with-PASID that request execute permission (such as for instruction fetches) and requests with supervisor privilege.

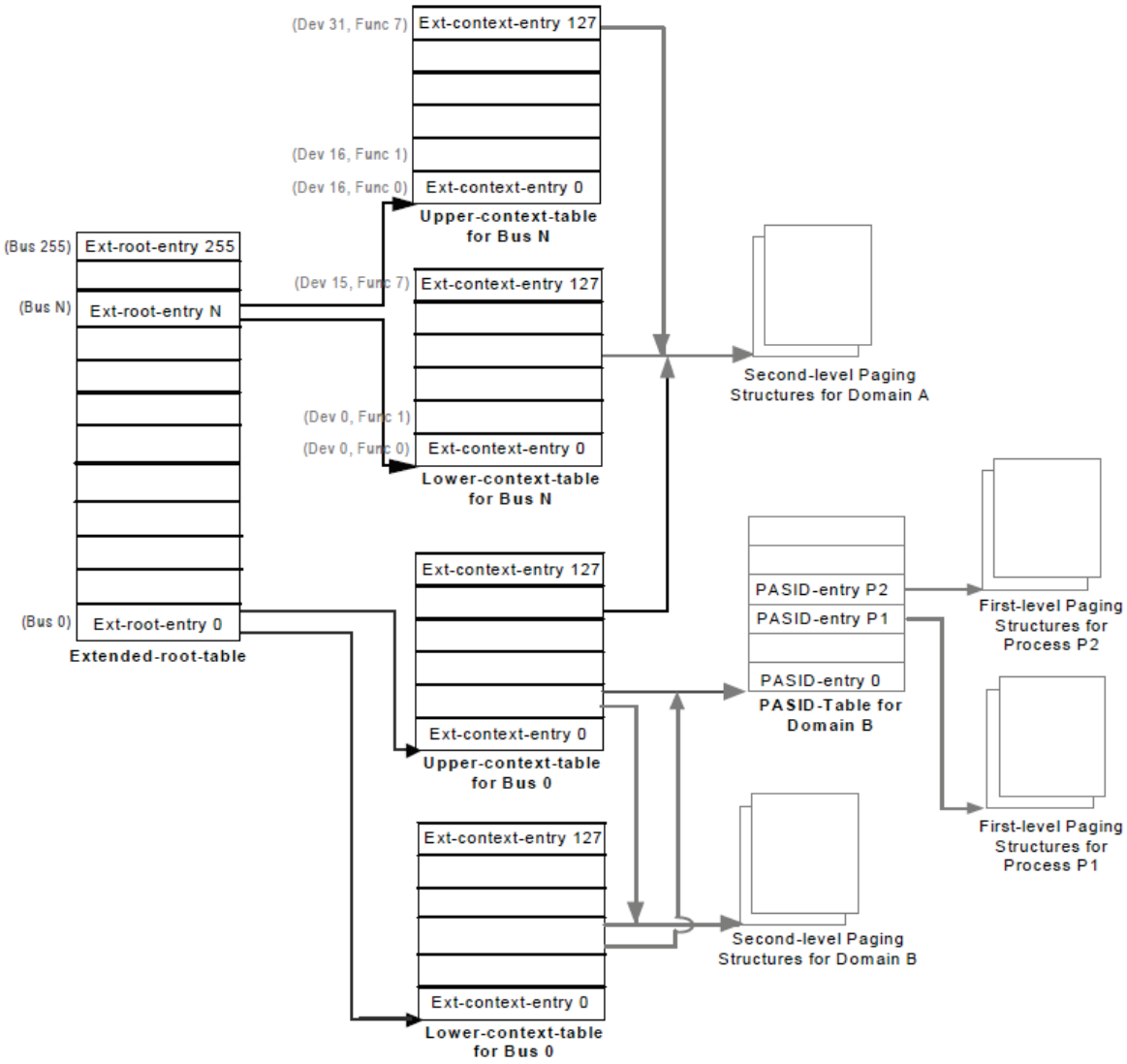

Figure 5 - Device to Domain Mapping Structures using Extended Root Table

System Firmware Responsibility

The UEFI system firmware is responsible for detecting the remapping hardware functions in the platform and reports the remapping hardware units in a platform to the system software through the DMA Remapping Reporting (DMAR) ACPI table. For detail ACPI table definition, please refer to [Intel VT-d] specification.

In the ACPI table, the first important data structure is the DMA Remapping Hardware Unit Definition (DRHD). Every DMA remapping unit should have a DRHD structure associated. The system should have at least one DRHD. Each DRHD may cover 1 or more PCI devices. For example, see below figure 8-31 from [Intel VT-d] specification. DRHD#1 has under its scope all devices downstream to the PCI-

Express root port located at (dev:func) of (14:0). DRHD#2 has under its scope all devices downstream to the PCI-Express root port located at (dev:func) of (14:1). DRHD#3 has under its scope a Root-Complex integrated endpoint device located at (dev:func) of (29:0). DRHD#4 has under its scope all other PCI compatible devices in the platform not explicitly under the scope of the other remapping hardware units. In this example, this includes the integrated device at (dev:func) at (30:0), and all the devices attached to the south bridge component.

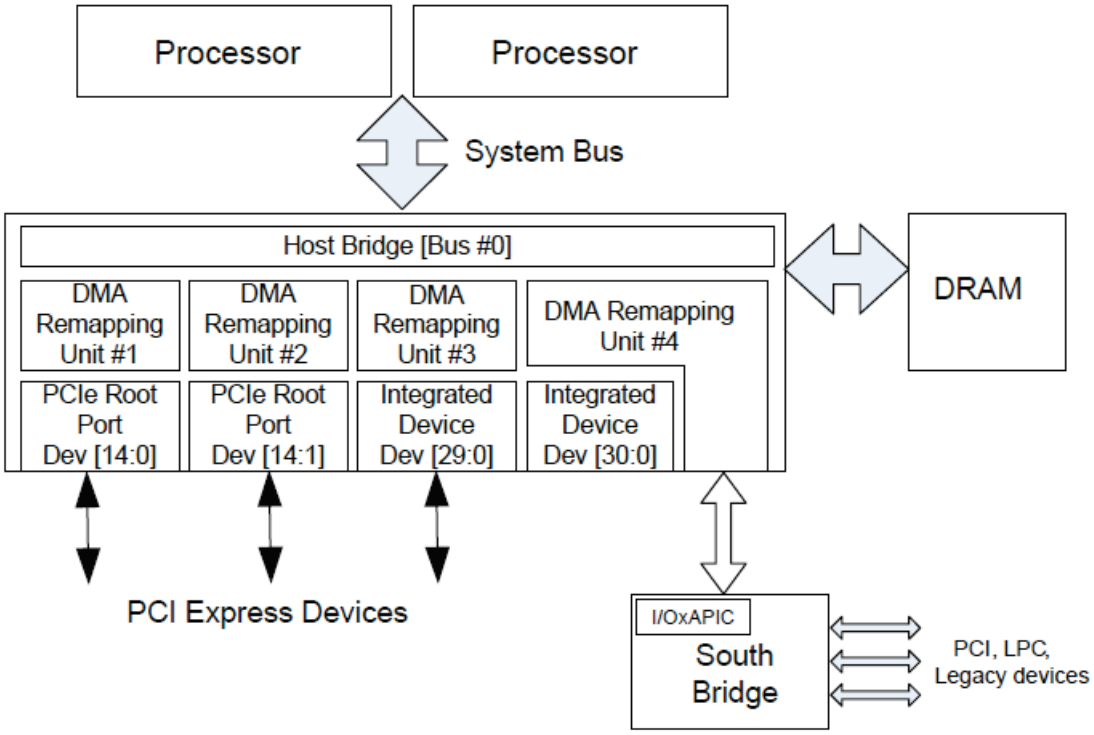

Figure 6 - Hypothetical Platform Configuration

When an OS parses this ACPI table, the OS can know which PCI device is managed by which DRHD unit. Then the OS will set the translation table for those devices only for one specific DRHD unit.

The second important data structure is the Reserved Memory Region Reporting (RMRR). A BIOS may report each such reserved memory region through the RMRR structures, along with the devices that requires access to the specified reserved memory region. Reserved memory ranges, which are either not DMA targets, or memory ranges that may be the target of the BIOS initiated DMA only during pre-boot phase (such as from a boot disk drive), must not be included in the reserved memory region reporting. The RMRR regions are expected to be used for the legacy usages requiring the reserved memory. In the current BIOS, most platforms report the reserved memory for the legacy USB, and the video buffer for the legacy video option rom for the integrated graphic device. Then the VMM/OS can setup an identical mapping DMA translation table for them during the system boot to make sure those devices still working.

The RMRR regions are expected to be used for legacy usages (such as USB, UMA Graphics, etc.) requiring the reserved memory. Platform designers should avoid or limit use of the reserved memory regions since these require system software to create holes in the DMA virtual address range available to system software and its drivers.

An ACPI Name-space Device Declaration (ANDD) structure uniquely represents an ACPI name-space

enumerated device capable of issuing DMA requests in the platform. A platform should report the ACPI Object Name, which matches the device name in the ASL code, and the ACPI Device Number, which matches the Enumeration ID of Device Scope in the DRHD table. As such, the ACPI OS can identify the source-id for this ACPI device. UEFI firmware uses a different method, EFI_DEVICE_PATH protocol, to identify the source-id for the ACPI device. Details of this implementation will be discussed later in the document.

In this paper, we will focus on the DRHD and RMRR usage in UEFI. The rest tables - Root Port ATS Capability Reporting (ATSR), Remapping Hardware Static Affinity (RHSA), are not discussed here. Please refer to [Intel VT-d] specification for more detail.

UEFI Support for DMA

PCI Bus for Bus Master Enabling

The UEFI specification defines the EFI_PCI_ROOT_BRIDGE_IO_PROTOCOL, which provides an IO abstraction for a PCI Root Bridge that is produced by a PCI Host Bus Controller. UEFI specification also defined the EFI_PCI_IO_PROTOCOL, which provides functions for the memory and I/O access on a PCI controller.

The EFI_PCI_IO_ATTRIBUTE_BUS_MASTER attribute is defined in EFI_PCI_IO_PROTOCOL. A PCI device driver may set this attribute to request to act as a bus master on the PCI bus. This operation is typically done in a UEFI driver model start function, such as EHCI, XHCI, ATA AHCI driver.

If the device driver does not require the bus master, the bus master should be disabled by default.

Example drivers are provided in EDK II open source:

- PCI: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/PciBusDxe
- PCI Root Bridge: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/PciHostBridgeDxe
- USB EHCI: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/EhciDxe
- USB XHCI: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/XhciDxe
- ATA AHCI: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Ata/AtaAtapiPassThru

PCI Bus for DMA

In order to support the DMA, the PCI_IO protocol defines a Map/Unmap() API. The Map() function provides the PCI controller–specific addresses needed to access the system memory. This function is used to map the system memory for the PCI bus master DMA accesses. For example, a PCI device driver (e.g. EHCI, XHCI, or AHCI) calls PCI_IO.Map() to input the system memory address and retrieve the device address out. Then the PCI device driver can program the device address to the hardware DMA register. When the DMA process is finished, the PCI device calls PCI_IO.Unmap() to release the corresponding resources.

The implementation of PCI_IO.Map/Unmap() forwards the request to PCI_ROOT_BRIDGE_IO.Map/Unmap() directly. The PCI Root Bridge driver is a platform specific driver. For example, on a traditional x86 platform, the PCI Root Bridge driver just return same address of host memory, because the x86 architecture can guarantee the DMA and the cache are consistent. An I/O agent can perform the direct memory access (DMA) to write-back memory and the cache protocol maintains cache coherency.

Support DMA Requests Above 4GiB

In some platforms, DMA is only supported below 4GiB memory address. However, the requested host memory may reside above 4GiB. This situation is not allowed using BusMasterCommonBuffer because the buffer

should be allocated by PCI_IO.AllocateBuffer, which guarantees the buffer is suitable for DMA. For BusMasterRead or BusMasterWrite, in Map() function, PCI Root Bridge may allocate below 4GiB memory, then sync content above 4GiB memory to a region below 4GiB memory before the BusMasterRead in Map(), or syncs the memory content below 4GiB to a region above 4GiB after BusMasterWrite in Unmap(). Finally the allocated below 4GiB memory will be freed in the Unmap() function.

IOMMU Protocol

EDK II introduces the EDK II_IOMMU_PROTOCOL to abstract IOMMU-based DMA access in UEFI. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/IoMmu.h)

The IOMMU protocol producer may consume ACPI table (such as Intel DMAR ACPI table [Intel VT-d], AMD IVRS ACPI table [AMD IOMMU], or ARM IORT ACPI table [ARM SMMU]) to get the IOMMU engine information.

Note there may be multiple IOMMU engines on one platform. For example, one for the integrated graphic device and one for the rest PCI devices (such as ATA/USB). All IOMMU engines are reported by one ACPI table. The IOMMU protocol provider should be based on the ACPI table. This single IOMMU protocol should handle multiple IOMMU engines.

This IOMMU protocol provider can use the UEFI device path to distinguish the location of the device, and find out corresponding IOMMU engine. The IOMMU protocol provides 2 set capabilities:

- SetAttribute() - Set the DMA Access Attribute, such as IOMMU_READ/IOMMU_WRITE control. As such, the UEFI firmware can do the fine granularity control on which memory can be accessed by which device.
- Map()/Unmap()/AllocateBuffer()/FreeBuffer() - Remap the DMA Buffer. These functions are similar to the ones defined in PCI_ROOT_BRIDGE_IO protocol. For example, to remap above 4GiB system memory address to below 4GiB device address. It provides AllocateBuffer/FreeBuffer/Map/Unmap for DMA memory. The remapping can be static (fixed at build time) or dynamic (allocated at runtime).

NOTE: The SetAttribute() is separated from the Map()/Unmap() function purposely, because the Map()/Unmap() does not provide the information on which device submits the DMA request. Without knowing which device submits the request, the IOMMU driver cannot provide the fine granularity control.

The IOMMU protocol consumer can be anyone who requests the DMA access. Take the PCI subsystem as example, the PciHostBridge driver should call IOMMU protocol AllocateBuffer()/FreeBuffer()/Map()/Unmap(), to get the mapped DMA buffer, if the IOMMU protocol is present.

The PciBus driver should call IOMMU protocol SetAttribute() to set the access attribute for the PCI device in Map/Ummap. Only after the access attribute is set, the PCI device can access the DMA buffer.

If a device is not in PCI subsystem such as the LPSS, the device driver should call same IOMMU protocol AllocateBuffer()/FreeBuffer()/Map()/Unmap() directly to get the mapped DMA buffer and call IOMMU protocol SetAttribute() to request the access control.

IOMMU Protocol for Non-Identical DMA Buffer

Current PCI_IO protocol supports non-identical DMA. For example, it supports the case that the host software is using above 4GiB memory but the 32bit device only supports below 4GiB memory. In the current implementation, an extra copy by the host CPU is needed to sync the memory between the below 4GiB and the above 4GiB. Also an error will be returned if the system fails to allocate below 4GiB memory.

This can be enhanced by setting a non-identical DMA buffer in the IOMMU engine. For example,

https://lists.01.org/pipermail/edk2-devel/2017-May/010360.html shows an ARM SMMU driver to remap DRAM for 32-bit PCI DMA statically in the IOMMU translation table.

IOMMU Protocol for Full Memory Encryption

IOMMU protocol can also be used in a full memory encryption solution, such as AMD memory encryption [AMD SME/SEV]. Because the DMA buffer cannot be encrypted, the IOMMU aware of the full memory encryption clears the encryption bit in page table for the DMA buffer before the device uses the DMA buffer.

For example, when IoMmu->Map() allocates a DMA buffer, Map() function clears the encryption bit. After that the DMA buffer can be used by a device. When IoMmu->Unmap() frees a DMA buffer, Unmap() function sets the encryption bit back. https://github.com/tianocore/edk2/tree/master/OvmfPkg/IoMmuDxe shows an AMD SEV IOMMU driver.

PI Support for DMA

Device Driver in PEI

The UEFI PI specification defines EFI_PEI_PCI_CFG2_PPI in PEI phase, which provides platform or chipset-specific access to the PCI configuration space for a specific PCI segment. But this PPI does not provide DMA support, such as Map/Unmap/AllocateBuffer/FreeBuffer.

The current implementation uses a lightweight PEI device driver. We can use the PCI USB controller as an example. In this case there is no PCI bus enumeration. The PCI BAR MMIO or IO resource is allocated statically and passed by PEI_USB_CONTROLLER_PPI (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Ppi/UsbController.h).

This PPI is produced by a silicon specific driver, (such as https://github.com/tianocore/edk2/tree/master/QuarkSocPkg/QuarkSouthCluster/Usb), and consumed by a generic USB controller driver (such as https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/XhciPei). As such the USB controller driver uses the MMIO allocated by the silicon driver and allocates the DMA buffer by the PEI memory services.

Additional PEIM examples from EDK II:

- EHCI: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/EhciPei
- SD: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/SdMmcPciHcPei
- UFS: https://github.com/tianocore/edk2/tree/master/MdeModulePkg/Bus/Pci/UfsPciHcPei

IOMMU PPI

In order to add DMA protection support in PEI, we introduced an EDK II_IOMMU_PPI to abstract IOMMU-based DMA access in PI PEI phase. (https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Ppi/IoMmu.h)

This IOMMU PPI is similar to the IOMMU protocol. The IOMMU PPI provides 2 set capabilities:

- SetAttribute() - Set the DMA Access Attribute, such as IOMMU_READ/IOMMU_WRITE control. This API is simpler than the protocol version, because it does not have DeviceHandle as the parameter. The reason is that PEI phase is so simple that there is no device handle concept.
- Map()/Unmap()/AllocateBuffer()/FreeBuffer() - Remap the DMA Buffer. The remapping can be static (fixed at build time) or dynamic (allocated at runtime). These APIs are same as the protocol version.

Now the USB controller driver must consume EDK II_IOMMU_PPI to allocate the DMA buffer, if the EDK II_IOMMU_PPI is published.

Device Driver in SMM

The PI specification defines EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL in SMM, which provides access to PCI I/O, memory and configuration space inside of SM. However,

the Map/Unmap/AllocateBuffer/FreeBuffer may return EFI_UNSUPPORTED.

Some existing PCI device drivers in SMM do not use the EFI_MM_PCI_ROOT_BRIDGE_IO_PROTOCOL, because it is not published on most platform. As such, a possible solution is to let the DXE or PEI agent driver allocate DMA buffer and pass the DMA buffer to SMM.

Using IOMMU Protocol in EDK II BIOS

Currently, most UEFI firmware exposes the IOMMU ACPI table to report IOMMU capabilities. It does not define a policy for the DMA protection. Current systems rely on the firmware for DMA protection in the boot process. This creates a need for firmware to set DMA protection policy for the boot device, and protect physical memory from unauthorized internal DMA and external DMA. Enabling the IOMMU with DMA remapping unit and using the IOMMU in firmware can meet this requirement.

This section describes how firmware can use IOMMU hardware to provide DMA protection, using Intel VT-d as the IOMMU example. The concept can be adopted for any IOMMU implementation.

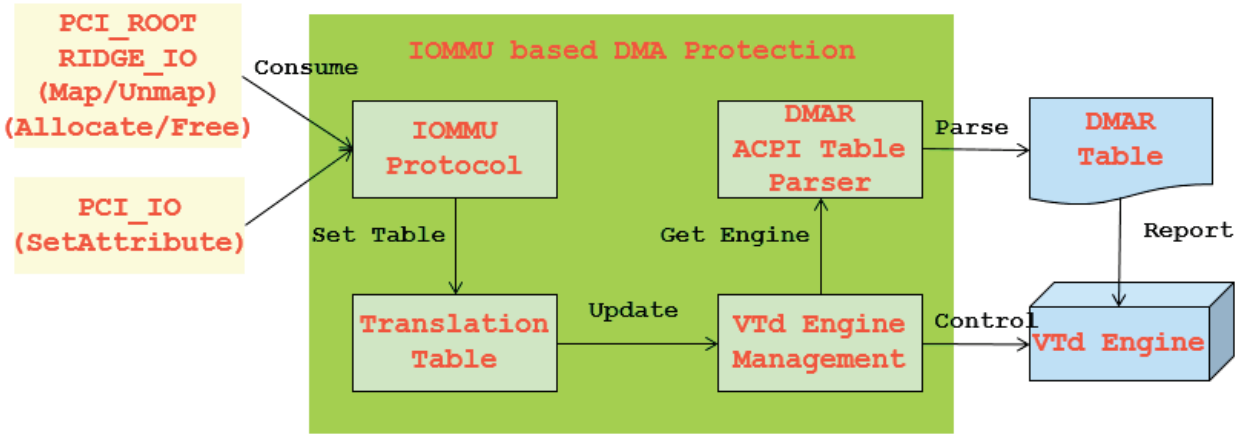

Figure 7 - IOMMU-based DMA Protection Component

The Intel VT-d driver (https://github.com/tianocore/edk2/tree/master/IntelSiliconPkg/Feature/VTd/IntelVTdDxe) consists of four major components:

1) DMAR ACPI table parser: parses the DMAR table to get the DRHD for the DMA remapping unit information and RMRR for reserved memory information.
2) Intel VT-d engine management: accessing the DMA remapping unit hardware register to enable or disable DMA remapping.
3) Translation table: Set up the DMAR root table, context table, and second level page table for specific PCI devices.
4) IOMMU Protocol: PCI device driver requests for DMA access via PCI_IO.Mapuse the IOMMU protocol. The Intel VT-d driver grants DMA access in the translation table. After the DMA transaction is finished, the PCI device driver calls PCI_IO.Unmap to free resources, then the Intel VT-d driver revokes DMA access in the translation table.

Step 0: Install IOMMU Protocol

The Intel VT-d driver installs the IOMMU protocol, which other device drivers consume for configuration (allocate/free the DMA buffer, set the DMA buffer attribute).

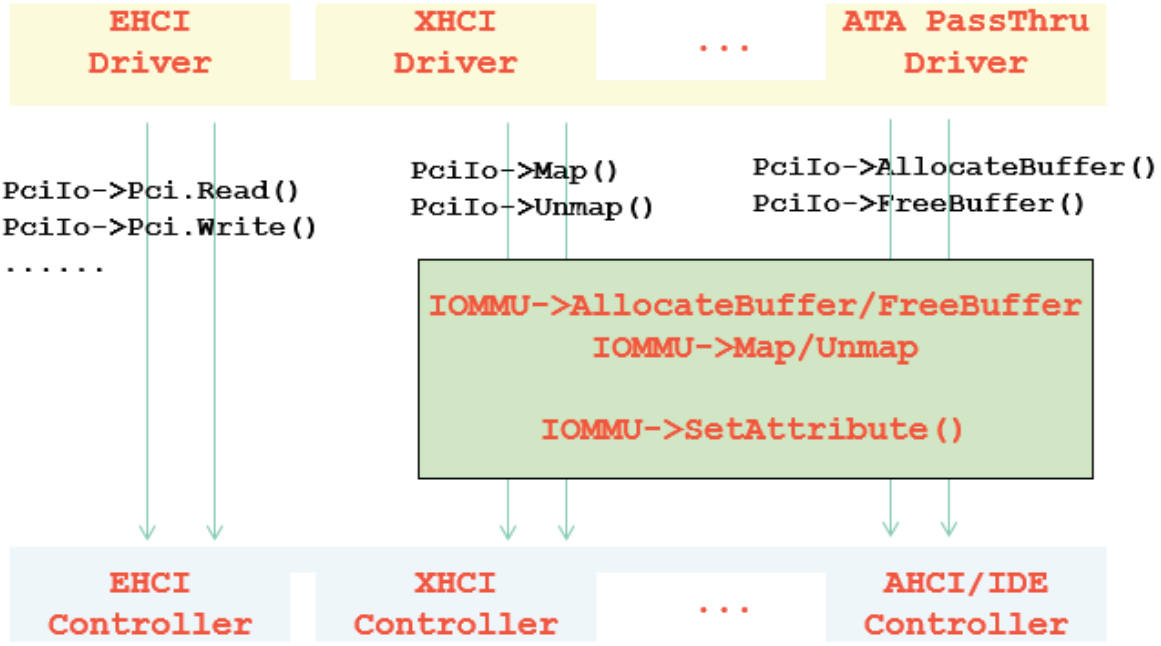

Figure 8 - DMA Protection Hook

If the IoMmu->SetAttribute() function is called before Intel VT-d is enabled, this request is cached by the Intel VT-d driver. This request will be processed later, when the translation table is setup.

Step 1: Parse DMAR ACPI Table

UEFI firmware reports DMA remapping capability via the ACPI DMAR table. In most platforms, this DMAR table is installed by a silicon driver (ex: https://github.com/tianocore/edk2-platforms/blob/devel-MinPlatform/Silicon/Intel/KabylakeSiliconPkg/SystemAgent/SaInit/Dxe/VTd.c). The DMAR table should be installed as soon as possible. On most platforms, the DMAR table can be installed at the PCI enumeration complete event (https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Protocol/PciEnumerationComplete.h). At least the DMAR table should be installed no later than the END_OF_DXE event (https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Guid/EventGroup.h).

The Intel VT-d driver registers a notification for the ACPI table event via the ACPI_SDT protocol. Once a platform installs the DMAR table, the Intel VT-d driver can access the ACPI table from the UEFI system configuration table.

First, the Intel VT-d driver gets the DRHD table from the DMA remapping unit information and saves the content in a driver global variable. If a DRHD entry does not set the INCLUDE_PCI_ALL flag, the Intel VT-d driver parses the DEVICE_SCOPE field and records PCI device information associated with this DMA remapping unit. If a DRHD entry sets the INCLUDE_PCI_ALL flag, the Intel VT-d driver scans the PCI bus and records all rest PCI devices information associated with this DMA remapping unit.

The Intel VT-d driver will not parse the RMRR table until the DMAR translation table is setup. Then the Intel VT-d driver gets the system reserved memory information, and always grants DMA access (read and write) to this memory to this specific device declared in the associated DEVICE_SCOPE of the RMRR table.

Step 2: Setup DMAR Translation Table

The Intel VT-d driver can setup the translation table, once it obtains the DMA remapping unit information.

The root-table and context-table should only set the valid entry for the PCI devices declared in the current DRHD. For those PCI devices not declared in DRHD, the root-table entry or context-table entry can be all zero. In order to maintain the fine granularity DMA buffer access control, each context-table entry points to a different second level page table and a unique domain-id is assigned to each second level page table. According to [Intel VT-d], each remapping unit in the platform should support as many number of domains as the maximum number of independently DMA-remappable devices expected to be attached behind it.

At same time, in order to save the memory usage for the second level page table, the second level page table is created on-demand. By default the translation table entry is not present.

If there is IoMmu->SetAttribute() request before the Intel VT-d is enabled, the request can be processed here.

Step 3: Get Platform VTD Policy

A platform may need provide a policy to the Intel VT-d driver. This work is done by EDKII_PLATFORM_VTD_POLICY_PROTOCOL (https://github.com/tianocore/edk2/blob/master/IntelSiliconPkg/Include/Protocol/PlatformVtdPolicy.h).

GetDeviceId(). This function provides the conversion from a UEFI device handle to an Intel VT-d device information (segment + source-id).

When the Intel VT-d driver updates the translation table, it must know the segment and source-id of the device requesting access. The segment is used to identify which DMA remapping unit manages the device. The source-id is used to identify the location of the context entry in the translation table of this DMA remapping unit. For an ACPI OS, the segment or source-id can be derived from the PCI device scope or the ANDD table. In UEFI firmware, this information is passed as a DeviceHandle in IoMmu->SetAttribute().

For the PCI device, it is easy to get PCI_IO protocol from the DeviceHandle, and get the Segment/Bus/Device/Function information from the PCI_IO protocol. Then the source-id can be constructed with Bus/Device/Function.

For the non-PCI device, there is not generic way to translate DeviceHandle to the segment and the source-id directly. As such, the platform must implement this function GetDeviceId() to return the segment and the source-id information, so that the Intel VT-d driver can find the corresponding DMA remapping unit and the context entry in the translation table.

The **GetExceptionDeviceList**() function returns a list of exception devices. The Intel VT-d driver always grants DMA access (read and write) to all system memory for these devices.

The IOMMU-based DMA protection solution relies on the PCI device driver implementation follows UEFI specification to call Map/Unmap/AllocateBuffer/FreeBuffer. If a device driver fails to follow this rule and accesses the memory directly, the DMA access is not granted and the device driver will get error from the hardware register.

In order to support such old device driver, a platform may choose to implement the GetExceptionDeviceList() to report these exception device. Care must be taken to make sure the exception devices are not tampered by a hacker.

A sample Intel VT-d platform policy is available for EDK II: https://github.com/tianocore/edk2/tree/master/IntelSiliconPkg/Feature/VTd/PlatformVTdSampleDxe.

Note this is a work-around for existing platforms that need to support older device drivers. We do not recommend that new platforms implement GetExceptionDeviceList() and directly return EFI_UNSUPPORTED.

Step 4: Enable DMA Remapping

After the DMAR translation table is setup properly, DMA remapping can be enabled. Each DMA

remapping unit has a set of standard remapping hardware registers. The Intel VT-d driver sets the translation enable (TE) bit in the Global Command Register to enable DMA remapping.

After that, the DMA access to system memory is not granted by default. The DMA access to the memory declared in RMRR is granted to the associated devices and the full DMA access is granted to the exception devices returned by GetExceptionDeviceList().

Step 5: Grant/Revoke DMA access rights in UEFI FIrmware

Once the Intel VT-d driver installs the IOMMU protocol, the IoMmu->SetAttribute() interface is called to update the translation table and grant or revoke DMA access. For BusMasterRead or BusMasterRead64, only read access is granted. For BusMasterWrite or BusMasterWrite64, only write access is granted. For BusMasterCommonBuffer or BusMasterCommonBuffer64, both read and the write access are granted.

Read and/or write access rights for certain memory can be granted for specific devices when PCI_IO.Map is invoked from the PCI device driver. IoMmu->SetAttribute() finds the second level page table entry according to the DMA buffer address. If the entry is not present, a new entry will be created. Then SetAttribute() will set the read/write access right to this page entry. If the DMA remapping unit does not have “Page-walk Coherency” capability, the page table update code must use CacheLineFlush to flush the system cache to the system memory. The page table update code must also flush the IOMMU context cache and/or the IOTLB to make the new entry take effect based upon the “Caching Mode” capability of the DMA remapping unit.

After the DMA buffer is used, the access right for certain memory must be revoked from certain device when PCI_IO.Unmap is invoked from the PCI device driver. IoMmu->SetAttribute() finds the second level page table entry according to the DMA buffer address and clear the read/write access right. The same page table update code should also flush the system cache, the IOMMU context cache and/or the IOTLB cache to make sure the update take effect immediately.

Step 6: Update DMA Remapping Status when Transferring Control to OS

If the OS does not manage the IOMMU, the DMA remapping unit should be turned off after the UEFI firmware transfer control to the OS. (Such as, DOS, legacy Windows, or legacy Linux).

If the OS can manage the IOMMU, the OS may choose to let the UEFI firmware keep the IOMMU enabled during boot. Before UEFI ExitBootService event, the UEFI firmware may update IOMMU translation table, because all IOMMU request goes there. After UEFI ExitBootService event, the UEFI driver never calls the IOMMU protocol. The OS may take over the control without disabling IOMMU and start to handle the IOMMU request from the OS driver. As such, the IOMMU mappings are preserved across the OS transition to provide the DMA protection.

Currently, IntelVTd driver uses a PcdVTdPolicyPropertyMask. (https://github.com/tianocore/edk2/blob/master/IntelSiliconPkg/IntelSiliconPkg.dec). BIT0 means to enable IOMMU during boot (If DMAR table is installed in DXE. If VTD_INFO_PPI is installed in PEI.) BIT1 means to enable IOMMU when transfer control to OS (ExitBootService in normal boot. EndOfPEI in S3) This PCD can be a static PCD set at build time, or a dynamic PCD set by platform at runtime. The default PCD (0x1) is the most compatible setting, allowing this capability to function with older, non-IOMMU aware OS and OS loaders.

One possible option, involving both a UEFI firmware and an IOMMU aware OS, is that the UEFI firmware may expose an interface to let the OS loader inform the UEFI firmware to ‘not’ tear-down IOMMU at ExitBootServices. The interface implementation may update the PcdVTdPolicyPropertyMask value, so that the IntelVTd driver can take appropriate action at the ExitBootService event.

Other options may be considered:

1) Define a new OsIndicator variable in UEFI specification. The UEFI OS can pass the IOMMU policy directly. This should be a UEFI secure boot like authenticated variable, so that the integrity is maintained.
2) Define additional services to have the pre-OS UEFI firmware tear-down earlier so that the OS can take over managing the IOMMU tables (or a combination of latter wherein the UEFI firmware exposes API's to manage the IOMMU in the pre-OS).

Using IOMMU PPI in EDK II

In existing UEFI firmware, the PEI device driver is needed for some special cases.

1) Debug output or communication: The debug device may be a USB device, which requires DMA access.
2) System recovery: The recovery image might be on a media device, such as USB, CDROM. In this case, the PEI storage driver accesses storage media to retrieve the image.
3) NAND flash access: A platform may put the DXE firmware volume on a NAND flash such as NVMe, UFS, eMMC, instead of SPI NOR flash. In this case, the PEI flash media driver reads the DXE firmware volume.
4) Early video: A platform may enable the graphic output in the early PEI phase to provide the better user experience.
5) Early capsule process: Some capsule require special handling in the PEI phase. For example, a platform may want to save capsule data on disk immediately after memory initialization.
6) Device access in S3 resume: If ACPI S3 resume requires device access, this driver may run in SMM or PEI.

In the cases described above, the firmware must setup DMA protection if the PEI device works as a bus mater and requires a DMA buffer. This section describes how firmware uses IOMMU hardware to provide DMA protection in PEI. This document uses Intel VT-d as the IOMMU example to explain proper implementation.

IOMMU-based DMA Protection in PEI

The Intel VT-d PMR PEIM driver (https://github.com/tianocore/edk2/tree/master/IntelSiliconPkg/Feature/VTd/IntelVTdPmrPei) is a lightweight version Intel VT-d driver. It is lightweight because it does not setup the Intel VT-d translation table, but uses the Protected Memory Region Register (PMR) to protect the system memory.

Intel VT-d capable hardware supports two set registers:

- Protected Low Memory Region Base/Limit (PLMR.Base/Limit)
- Protected High Memory Region Base/Limit (PHMR.Base/Limit)

If a region is within [Base, Limit], then this region is DMA protected.

The VTdDxe driver depends on DMAR ACPI table. However, this ACPI table is not installed in PEI phase. VTD_INFO_PPI is defined so the PEI driver can obtain Intel VT-d information (https://github.com/tianocore/edk2/blob/master/IntelSiliconPkg/Include/Ppi/VtdInfo.h). This PPI is the DMAR table reporting in PEI phase. The VTdPmr PEIM depends on VTD_INFO_PPI.

A sample Intel VT-d info PEIM is available in EDK II: https://github.com/tianocore/edk2/tree/master/IntelSiliconPkg/Feature/VTd/PlatformVTdInfoSamplePei.

During initialization, the VTdPmr PEIM only allocates one big DMA memory. It sets PMLR for the memory below this DMA memory, and PHMR for the memory above this DMA memory. As such, only this big DMA memory can be used for the device DMA.

During runtime, the VTdPmr PEIM manages the DMA memory. When the VTdPmr receives the common buffer allocation request, it allocates the common DMA buffer from top of the DMA memory. When the VTdPmr receives the read/write buffer map request, it allocates the

read/write DMA buffer from bottom of the DMA memory.

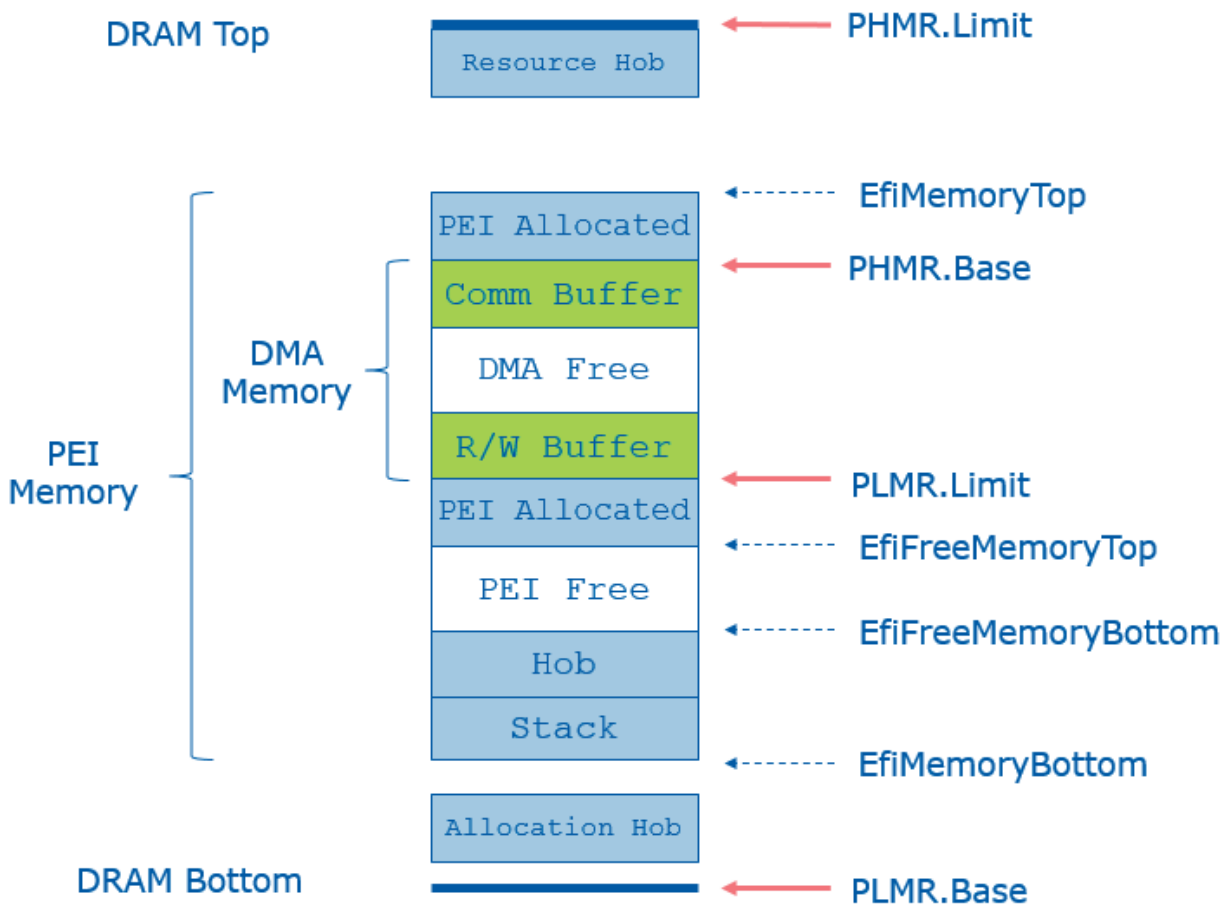

Figure 9 - Memory Map Example

DMA Protection at the End of PEI

In normal boot, PEI DxeIpl launches DxeCore and transfers control to DXE. There is no need to turn off DMA protection if the DXE IOMMU driver has knowledge of the IOMMU. For example, the VTdDxe driver sets up the Intel VT-d translation table, then disables PMR. As a result the VTdPmr PEIM does not disable PMR and the PMR protection exists at the early of DXE phase.

For ACPI S3 resume, PEI transfers control to the OS. In this case, the VTdPmr driver consumes the PcdVTdPolicyPropertyMask. (https://github.com/tianocore/edk2/blob/master/IntelSiliconPkg/IntelSiliconPkg.dec) to determine if there is need to turn off the IOMMU protection at the EndOfPei event.

The PCI BME requirement is same for the PEI S3 resume. Before the system transfers control to OS, the PCI BME bit should be off. If a driver enables bus master for some function, it should disable bus master at EndOfPei.

Miscellaneous Topics

PCI Bus Master Enable

The PCI BME bit in a PCI device should not be enabled by default. Only the device driver should enable PCI BME.

The PCI BME bit in a PCI bridge should only be enabled when there is a device under the bridge. If there is not device under a PCI bridge, the BME bit should be disabled.

Some systems do not support Intel VT-d. In this case (or whenever Intel VT-d mitigation is not feasible), some DMA protection is possible through disabling bus mastering in the root bridge up through the point in which locks are applied and SMM interfaces that are not intended to be exposed to potentially malicious code are unregistered. Disabling bus mastering in the endpoint device is possible, but the device cannot be trusted to honor this setting. Instead this mitigation relies on the behavior of root bridges, which should drop DMA transactions when bus mastering is disabled. The examples below configure root bridges to have bus mastering disabled up through EndOfDxe.

DMA Memory Alignment

There might be a basic alignment requirement in an IOMMU. For example, Intel VT-d requires the DMA buffer to be 4KiB aligned, because the translation table uses 4KiB blocks. However, when the PCI device driver submits the request by PCI_IO.Map, it does not consider any alignment requirements. In this case, the IOMMU driver remaps DMA to 4KiB aligned memory.

Non-PCI Device Support

As previously discussed, EDKII_PLATFORM_VTD_POLICY_PROTOCOL.GetDeviceId() must be implemented for non-PCI devices. In this case, the Intel VT-d driver can obtain the segment to identify the DMA remapping unit and source-id to identify the location of the context entry in the DMA remapping unit translation table.

UEFI Drivers not Following the UEFI Specification

As previously discussed, a platform may return a list of exception devices using the function EDKII_PLATFORM_VTD_POLICY_PROTOCOL.GetExceptionDeviceList(). The exception list contains device drivers that does not follow the UEFI Specification to access the system memory via DMA buffer.

This method is only a work-around for older platform who wants to support old device driver. We do recommend newer platforms implement the GetExceptionDeviceList() function, and port the function to directly return EFI_UNSUPPORTED.

DMA Access Before IOMMU is Enabled

The DMA remapping unit information is reported via a DMAR ACPI table. The Intel VT-d driver cannot work before the silicon initializes the Intel VT-d and reports the DMAR table.

A platform should disable DMA access by default before the IOMMU is enabled. A platform may keep PCI BME disabled, or setup Intel VT-d PMR register.

The only exception could be a debug device, such as the USB debug port. This device driver sets a DMA access request when the IOMMU protocol is installed, even before the IOMMU is enabled. Now DMA access can be granted automatically when the IOMMU is enabled.

For server RAS system, the platform BIOS should enable hardware mechanism to prevent rogue DMA attacking hot plugged CPU. [DMA7]

Device Driver in SMM

Most UEFI firmware installs device drivers in SMM that require the DMA buffer to be allocated. Some examples are described below:

LegacyUsb: this driver is expected to report the DMA buffer in reserved memory via RMRR table. In this case, the Intel VT-d driver always grants access to the RMRR buffer for the USB device.

OpalPassword: this driver allocates the DMA buffer in reserved memory, then calls IoMmu->Map() and IoMmu->SetAttribute() to grant access rights for the DMA buffer once the device handle is installed.

DebugCommunicationLibUsb3: the DXE instance registers a protocol notification for IOMMU and PCI_IO. Once PCI_IO is installed for the USB

controller, this module calls IoMmu->Map() and IoMmu->SetAttribute() to grant access rights for the allocated DMA buffer.

Device Driver in PEI

As previously discussed, DMA may be enabled for a boot path where a PEI device driver is dispatched (ex: XhciPeim, PeiGraphic, DebugCommunicationLibUsb3, OpalPasswordSmm).

XhciPeim: this PEIM requires an IOMMU call to allocate the DMA buffer. Even though this an EDK II implementation, note that IOMMU_PPI may not exist on all platforms. The PEI device driver may attempt to locate IOMMU_PPI to obtain a DMA buffer, ten use PEI memory services to allocate the DMA buffer if the PPI does not exist.

PeiGraphic: because the graphics UMA region resides at a fixed address, the system reports the UMA region with RMRR in VTD_INFO_PPI. In this case, the VTdPmr driver grants access to the UMA buffer.

DebugCommunicationLibUsb3: the PEI instance registers a PPI notification for IOMMU PPI. Once the IOMMU_PPI is installed, this module calls IOMMU to reallocate the DMA buffer.

OpalPasswordSmm: In the S3 resume path, the boot script should not enable BME by default. The boot script may dispatch device drivers, such as the OpalPassword SMM driver, which uses DMA to send an OPAL command to unblock the OPAL device. A PEI stub calls IOMMU to allocate the DMA buffer, allowing the SMM device driver to consume the new DMA buffer and complete the operation. The OpalPassword driver disables BME before handing off to the OS.

DMA Access When Transferring Control to the OS

This document previously discussed several options for controlling the VTd state when UEFI firmware transfers the control to the OS, such as at ExitBootServices or EndOfPei. A platform may set policy as needed.

Compatibility Support Module (CSM)

The CSM installs a legacy BIOS support layer on UEFI firmware. There are some special memory requirements for legacy compatibility that need to be considered. For example

- Some PCI legacy option ROMs use the Extended BIOS Data Area (EBDA)
- Some option ROMs use memory below 1MiB (“Low PMM”)
- Some option ROMs use memory below 16MiB (“High PMM”)
- Some PCI legacy option ROMs allocate “free memory” by searching for zeroed memory blocks between 0x60000 ~ 0x88000 and assuming the region is not in use.

The Intel VT-d driver does not have knowledge of memory regions used by PCI legacy option ROMs. One method for maintaining compatibility is to let the CSM driver grant direct access to specific “legacy memory regions” for devices associated with a legacy option ROM.

DMA Error Debug

If UEFI firmware fails to function after enabling IOMMU protection, this indicates DMA protection is not enabled correctly. Possible areas to debug:

1) Device Driver:
   a) Firmware does not update the PciHostBridge driver or the PciBus driver to consume the IOMMU protocol.
   b) Firmware does not update the non-PCI device driver to consume the IOMMU protocol.
   c) Firmware does not update the SMM device driver to consume the IOMMU protocol.
   d) Firmware does not update the debug device driver to consume the IOMMU protocol.
   e) Firmware does not update the CSM driver to consume the IOMMU protocol.
   f) Firmware does not update the PEI device driver to consume the IOMMU PPI.
2) Platform Code:
   a) Does not implement PLATFORM_VTD_POLICY_PROTOCOL.GetDeviceId() for non-PCI devices.
   3) IOMMU:
   a) Intel VT-d driver does not correctly configure the context table or second level page table.
   b) Intel VT-d driver does not flush the system memory cache of the context table or the second level page table.
   c) Intel VT-d driver does not flush the IOMMU context cache or the IOTLB cache.

The DMA remapping unit has the capability to generate an error log for DMA errors. This error information is dumped as debug message by the Intel VT-d driver. Example:

```
==================================
  FRCD_REG[0] - 0x8000000C000000B8 0000000089AF1000
    Fault Info - 0x0000000089AF1000
    Source - B00 D17 F00
    Type - 0 (write)
    Reason - C
==================================
```

- "Fault Info" indicates the DMA buffer address where causes the fault.
- "Source" indicates the PCI bus/device/function number
- "Type" indicates the write or *read error.*
- *"Reason" i*ndicates the fault reason. See Appendix A of [V
- The Intel VT-d driver also dumps the full DMAR table. This allows the plT-d]. atform owner to check the DMA remapping unit information.

Performance

If an IOMMU protocol is produced, each PciIo->Map()/UnMap() call invokes IoMmu->SetAttribute(). The later call updates the Intel VT-d translation table. The final performance impact depends on two factors:

1) How many IoMmu->SetAttribute() calls are invoked? (AccessCount)
2) How long is each IoMmu->SetAttribute() call? (TimeOfUpdate)

For the first factor (AccessCount), we collected data from a code-name "Kaby Lake" reference platform booting to Microsoft* Windows 10:

```
==================================
engine [0] access
  PCI B00 D02 F00 - 1
engine [1] access
  PCI B00 D00 F00 - 0
  PCI B00 D05 F00 - 0
  PCI B00 D08 F00 - 0
  PCI B00 D13 F00 - 0
  PCI B00 D14 F00 - 181
  PCI B00 D14 F02 - 0
  PCI B00 D14 F03 - 0
  PCI B00 D15 F00 - 0
  PCI B00 D15 F01 - 0
  PCI B00 D15 F02 - 0
  PCI B00 D15 F03 - 0
  PCI B00 D16 F00 - 0
  PCI B00 D17 F00 - 713
  PCI B00 D19 F00 - 0
  PCI B00 D19 F01 - 0
  PCI B00 D1E F00 - 0
  PCI B00 D1E F01 - 0
  PCI B00 D1E F02 - 0
  PCI B00 D1E F03 - 0
  PCI B00 D1E F04 - 0
  PCI B00 D1E F06 - 0
  PCI B00 D1F F00 - 0
  PCI B00 D1F F02 - 0
  PCI B00 D1F F03 - 0
  PCI B00 D1F F04 - 0
  PCI B00 D1F F06 - 0
  PCI BF0 D1F F00 - 0
  PCI B00 D19 F02 – 0
==================================
```

The data shows the video access count is 1, the USB access count is 181, and the ATA access count is 713. These values are as expected, based on device behavior:

- The RMRR for the video device is set once.
- The USB keyboard driver periodically checks for a key.
- The OS loader is loaded from an ATA disk, then loads the OS kernel from the ATA disk.

Performance log for the second measurement (TimeOfUpdate) was collected on a code-name "Kaby Lake" reference system booting to the UEFI Shell:

```
================================================================================
...
==[ Cumulative ]========
(Times in microsec.)       Cumulative    Average     Shortest    Longest
   Name           Count     Duration     Duration    Duration    Duration
--------------------------------------------------------------------------------
...
S0000B00D14F0      682         674          0           0          53
S0000B00D17F0      239         694          2           0         168
================================================================================
```

This data shows the average time of IoMmu->SetAttribue() is about 1~2 microsecond, which has minimal impact on system performance.

Conclusion

The IOMMU is an attractive feature for protecting UEFI firmware against DMA attacks. This paper describes methods to enable the IOMMU and protect DMA in the UEFI pre-boot phase.

Platform firmware sets the DMA policy based upon the threat model:

A. PCI BME may be sufficient for external PCI devices.
B. IOMMU may be required for all internal/external PCI/non-PCI devices.

Call to Action

If the IOMMU-based DMA protection is required, then the following actions are recommended:

1) For device drivers:
   a) PCI OpROM & device driver code should follow the UEFI Specification and call PCI_IO Map/Unmap/AllocateBuffer/FreeBuffer for the DMA buffer.
   b) Non-PCI device drivers should call the IOMMU protocol Map/Unmap/AllocateBuffer/FreeBuffer and SetAttribute for the DMA buffer.
   c) The SMM device driver should call IOMMU protocol Map and SetAttribute for DMA buffer.
   d) The debug device driver should call IOMMU protocol Map and SetAttribute for DMA buffer.
   e) The UEFI device driver should disable bus master and put controller to halt state in ExitBootServices. [UEFI DWG]
   f) The PEI device driver should call IOMMU PPI for DMA buffer.
2) For silicon driver:
   a) The IOMMU driver should implement IOMMU_PROTOCOL.SetAttribute() to provide fine granularity control for DMA buffer access.
   b) The silicon code should expose the IOMMU related ACPI table as early as possible. Ideally at PciEnumerationComplete protocol callback, or at least no later than EndOfDxe event.
   c) The silicon code should initialize IOMMU in the PEI phase, if the PEI IOMMU is required.
   d) The silicon code or the platform code may expose IOMMU information PPI in the PEI phase, such as VTD_INFO_PPI.

e) The silicon code should disable the PEI device controller at EndOfPei.

3) For platform code:
   a) UEFI firmware should use the IOMMU to resist DMA attack.
   b) UEFI firmware should make sure that the DMA is disabled before IOMMU is enabled.
   c) Platform code should expose PLATFORM_VTD_POLICY_PROTOCOL.GetDeviceId() for non PCI device.
   d) Platform code should not implement PLATFORM_VTD_POLICY_PROTOCOL.GetExceptionDeviceList().
   e) Platform firmware sets IOMMU policy, such as PcdVTdPolicyPropertyMask. Policy should be set in both PEI and DXE phases.

Acknowledgement

After publishing the first version of this document, we received feedback on IOMMU design, and using IOMMU for platform hardening. We would like to thank Corey Kallenberg (Apple), Leo Duran (AMD), Brijesh Singh (AMD), Ard Biesheuvel (Linaro), Jeramiah Cox (Microsoft), Sean Brogan (Microsoft), Nathaniel L Desimone (Intel), Jenny Huang (Intel), Sugumar Govindarajan (Intel), Michael D Kinney (Intel), Ruiyu Ni (Intel), John Loucaides (Intel), and Dmytro Oleksiuk (Cr4sh).

Glossary

ACPI – Advanced Configuration and Power Interface. The specification defines a new interface to the system board that enables the operating system to implement operating system-directed power management and system configuration.

DMA – Direct Memory Access.

DMAR – Intel DMA Remapping Reporting table. It is the ACPI table on DMA remapping hardware units in a platform.

DRHD – Intel DMA Remapping Hardware Unit Definition. It is the structure in DMAR table, which represents a remapping hardware unit present in the platform.

Intel VT-d – Intel® Virtualization Technology for Directed I/O.

IOMMU – IO Memory Management Unit.

IORT – ARM IO Remapping Table in ACPI.

IVRS – AMD IO Virtualization Reporting Structure in ACPI.

PASID – Process Address Space ID, in conjunction with the Requester ID, uniquely identifies the address space associated with a transaction.

PI – Platform Initialization. Volume 1-5 of the UEFI PI specifications.

PMR – Protected Memory Region. It is one feature in Intel VTd.

RMRR – Intel Reserved Memory Range Reporting. It is the structure in DMAR table, which reports BIOS allocated reserved memory ranges that may be DMA targets.

UEFI – Unified Extensible Firmware Interface. Firmware interface between the platform and the operating system. Predominate interfaces are in the boot services (BS) or pre-OS. Few runtime (RT) services.

References

[AMD IOMMU] AMD I/O Virtualization Technology, https://support.amd.com/TechDocs/48882_IOMMU.pdf

[AMD SME/SEV] AMD Memory Encryption, http://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[ARM SMMU] ARM System Memory Management Unit, https://developer.arm.com/products/system-ip/system-controllers/system-memory-management-unit

[ARM IORT] IO Remapping Table – Arm, https://static.docs.arm.com/den0049/b/DEN0049B_IO_Remapping_Table.pdf

[BusMastering] Bus Mastering:
http://www.pcguide.com/ref/mbsys/buses/func_Mastering.htm
http://www.pcguide.com/ref/mbsys/buses/types/pciMastering-c.html

[DMA1] Forristal, Hardware Involved Software Attacks, Dec 2011, http://forristal.com/material/Forristal_Hardware_Involved_Software_Attacks.pdf

[DMA2] Sang, Nicomette and Deswarte, I/O Attacks in Intel-PC Architectures and Countermeasures, 2011
http://www.syssec-project.eu/media/page-media/23/syssec2011-s1.4-sang.pdf

[DMA3] Aumaitre and Devine, Subverting Windows 7 x64 Kernel with DMA attacks, 2010
https://conference.hitb.org/hitbsecconf2010ams/materials/D2T2%20-%20Devine%20&%20Aumaitre%20-%20Subverting%20Windows%207%20x64%20Kernel%20with%20DMA%20Attacks.pdf

[DMA4] Russ Sevinsky, Funderbolt - Adventures in Thunderbolt DMA Attacks, 2013, https://media.blackhat.com/us-13/US-13-Sevinsky-Funderbolt-Adventures-in-Thunderbolt-DMA-Attacks-Slides.pdf

[DMA5] :

- Ulf Frisk, DMA attacking over USB-C and Thunderbolt 3, 2016, http://blog.frizk.net/2016/10/dma-attacking-over-usb-c-and.html
- macOS FileVault2 Password Retrieval, 2016, http://blog.frizk.net/2016/12/filevault-password-retrieval.html
- Attackin UEFI and Linux, 2017, http://blog.frizk.net/2017/01/attacking-uefi-and-linux.html

[DMA6] Alex Ionescu, Getting Physical With USB Type-C - Windows 10 RAM Forensics and UEFI Attacks, http://alex-ionescu.com/publications/Recon/recon2017-bru.pdf

[DMA7] Cuauhtemoc Chavez-Corona, et al, Abusing CPU Hot-Add weaknesses to escalate privileges in Server Datacenters, https://cansecwest.com/slides/2017/CSW2017_Cuauhtemoc-Rene_CPU_Hot-Add_flow.pdf

[EDK2] UEFI Developer Kit
https://github.com/tianocore/tianocore.github.io/wiki/Driver-Developer

[HSTI] Hardware Security Testability Specification
https://msdn.microsoft.com/en-us/library/windows/hardware/mt712332.aspx

[PCI] PCI Local Bus Specification, Revision 3.0, https://pcisig.com/specifications

[PCIExpress] PCI Express Base Specification, Revision 3.1, https://pcisig.com/specifications

[TXT] Intel TXT Software Development Guide,
https://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-txt-software-development-guide.pdf

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, http://www.uefi.org/specifications

[UEFI Book] Zimmer,, et al, "Beyond BIOS: Developing with the Unified Extensible Firmware Interface," 2nd edition, Intel Press, January 2011

[UEFI DWG] UEFI Driver Writer's Guide, https://github.com/tianocore/tianocore.github.io/wiki/UEFI-Driver-Writer%27s-Guide

[UEFI Overview] Zimmer, Rothman, Hale, "UEFI: From Reset Vector to Operating System," Chapter 3 of *Hardware-Dependent Software*, Springer, February 2009

[UEFI PI Specification] UEFI Platform Initialization (PI) Specification, volumes 1-5, http://www.uefi.org/specifications

[Intel VT-d] Intel Virtualization Technology for Directed I/O specification, https://www-ssl.intel.com/content/www/us/en/embedded/technology/virtualization/vt-directed-io-spec.html?wapkw=virtualization+technology+for+directed+i%2fo+specification

[WindowsRequirement] Hardware Compatibility Specification for Systems for Windows 10, https://docs.microsoft.com/en-us/windows-hardware/design/compatibility/systems

[WindowsDMA] Blocking the SBP-2 driver and Thunderbolt controllers to reduce 1394 DMA and Thunderbolt DMA threats to BitLocker http://support.microsoft.com/kb/2516445

About the Authors

Jiewen Yao is an EDK II BIOS architect at Intel Corporation.

Vincent J. Zimmer is a Senior Principal Engineer at Intel Corporation.

Star Zeng is an EDK II BIOS architect at Intel Corporation.