# *White Paper*

# *A Tour beyond BIOS Using Intel ® VT-d for DMA Protection in UEFI BIOS*

*Jiewen Yao*
*Intel Corporation*

*Vincent J. Zimmer*
*Intel Corporation*

January 2015

# Executive Summary

This paper presents on a design methodology for using Intel VT-d in a UEFI BIOS for purposes of resisting DMA attacks against the host UEFI firmware from devices.

**Prerequisite**
This paper assumes that audience has EDKII/UEFI firmware development experience. This paper also assumes that the audience has basic knowledge of PCI and DMA.

# Table of Contents

# Overview

## Introduction to VT-d

In the PC world, DMA (Direct Memory Access) attack is considered as one hardware related attack [DMA1] [DMA2] [DMA3]. It can bypass any software check and write to system memory directly. OS vendors like Microsoft were aware those issues and providing some advocacy to address a mitigation, such [DMA4], and even promoted this concern to be a requirement in security reporting interfaces [HSTI] and tests [WHCK System].

In terms of a more generic taxonomy, paging and virtualization hardware within the CPU for host-based execution isolation are referred to as a the Memory-Management Unit (MMU), and hardware within is for isolation of I/O devices is referred to as the Input/Output Memory Management Unit (IOMMU).

Intel Virtualization Technology for Direct I/O [VT-d] is designed for a Virtual Machine Monitor (VMM), to support I/O virtualization. Since VT-d has the capability for access control, it provides one possible solution to block a DMA attack.

## Introduction to the EDKII

The UEFI specification [UEFI] is adopted in most server, PC, mobile and tablet device designs. EDKII is an open source implementation for UEFI. The EDKII project has an implementation of the UEFI Platform Initialization (PI) Driver Execution Environment (DXE) to produce UEFI main specification prescribed interfaces. The DXE core, likes an OS kernel, has a dispatcher and scheduler to run other drivers, and the DXE core also produces services and functions that architecturally required by UEFI. In the DXE phase, some drivers are responsible for system integrity. For example, the DXE security architecture protocol (SAP) 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 that need to be protected.

**Summary**
This section provided an overview of Intel VT-d and EDKII.

# Goal and Motivation

Most UEFI BIOS implementations for Intel platforms report a VT-d-related ACPI table to expose the VT-d capability of the platform. However, most UEFI BIOS's today only reports the capability and do not enable any DMA remapping unit for protection of the DXE core and DXE/UEFI drivers.

In this paper, we will present one possibility methodology on how to enable the DMA remapping unit in UEFI to prevent DMA attacks against UEFI services and functions. This allows for the critical drivers to be protected and integrity maintained in UEFI BIOS throughout the boot.

**Summary**
This section provided the goals and motivation for using VT-d in UEFI.

# Intel VT-d introduction

## VT and VT-d

Intel Virtualization Technology (VT) is a processor feature that enables the concurrent execution of 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 (VT-d) is for IO virtualization, which provides capability of

- *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.

VT-d and VT are different features. They can be enabled independently from technical perspective. A solution may activate VT only without VT-d, if there is no DMA concern. Or a solution may active VT-d only without VT, if there is no need for system partitioning, or separation of host-based execution.

For example, the OS can also use VT-d on DMA remapping feature for below purposes:

- *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.

In this paper, we will focus on the DMA remapping feature only. For more detailed information regarding features of VT-d, such as Interrupt remapping or Interrupt posting, please refer to [VT-d] specification.

## DMA remapping

The key concept of DMA remapping is address translation. See below figure 3-5 from the [VT-d] specification. The left hand side is for processor virtualization, and the right hand side is for IO virtualization. On the right hand side, both device 1 and device 2 want to access 0x4000 memory address. The DMA remapping unit (DMA memory management) can map guest physical address (GPA) to host physical address (HPA). As final result, device 1 access host physical address 0x6000 and device 2 accesses host physical address 0x3000.

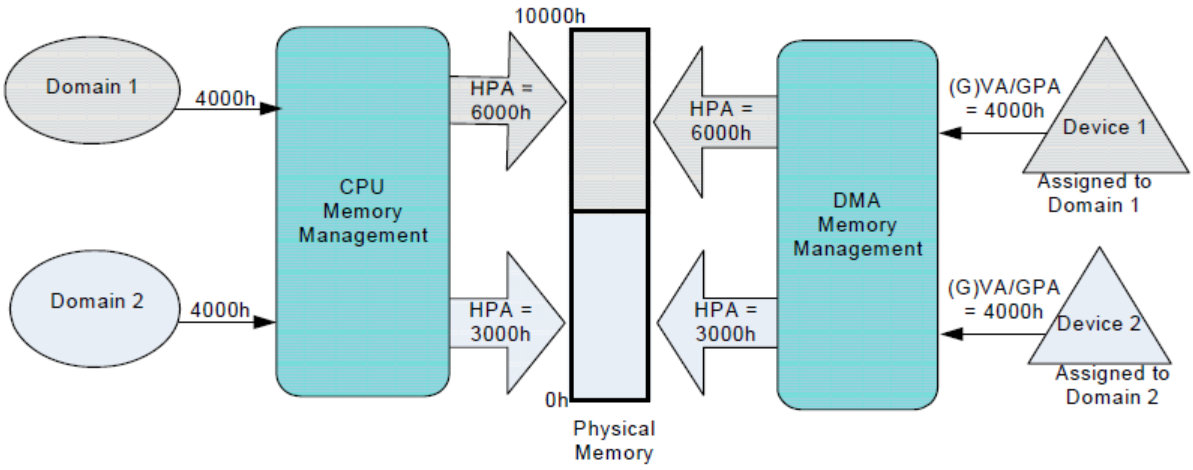



**Figure 3-5. DMA Address Translation**

## Address Translation Structures

In order to let devices find the proper host physical address, the DMA remapping unit needs to be set up with a "translation table" for each device.

Initially, the system has its **root-table** functions as the top level structure to map devices to their respective domains. The location of the root-table is in the VT-d register named **Root Table Address Register**. The root table contains 256 root-entries to cover the PCI bus number space (0-255). Each **root-entry** contains a **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. So for each PCI device (with bus number, device number and function number), there is an address translation structure (second level page-table) associated. See below figure 3-7 from [VT-d] specification.

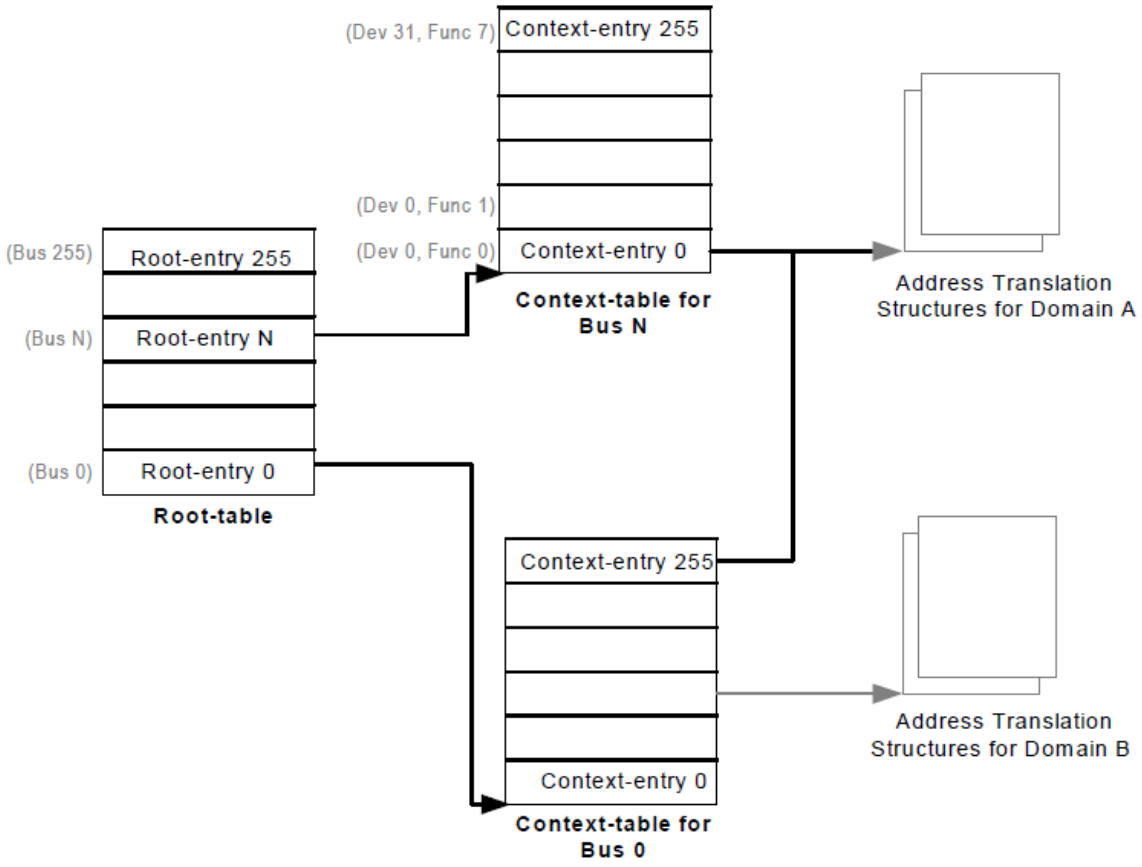



**Figure 3-7. Device to Domain Mapping Structures using Root-Table**

Next, the address translation structure for the **second level page table** is similar to the page table of CPU. The difference is that second level page table of VT-d uses X (execution), W (write), R (read) bits, whereas CPU page tables uses XD (executable), Read/Write(RW), P(Present) bit for execution, write and read access control. So the entry names are changed to **SL-PML4E**, **SL-PDPE**, **SL-PDE**, and **SL-PTE**. See below figure 3-9 and 3-39 from [VT-d] specification.

The full translation process is:

1) Parse root-table and context table (Illustrated in Figure 3-7)
2) Parse second level page table. (Illustrated Figure 3-9)

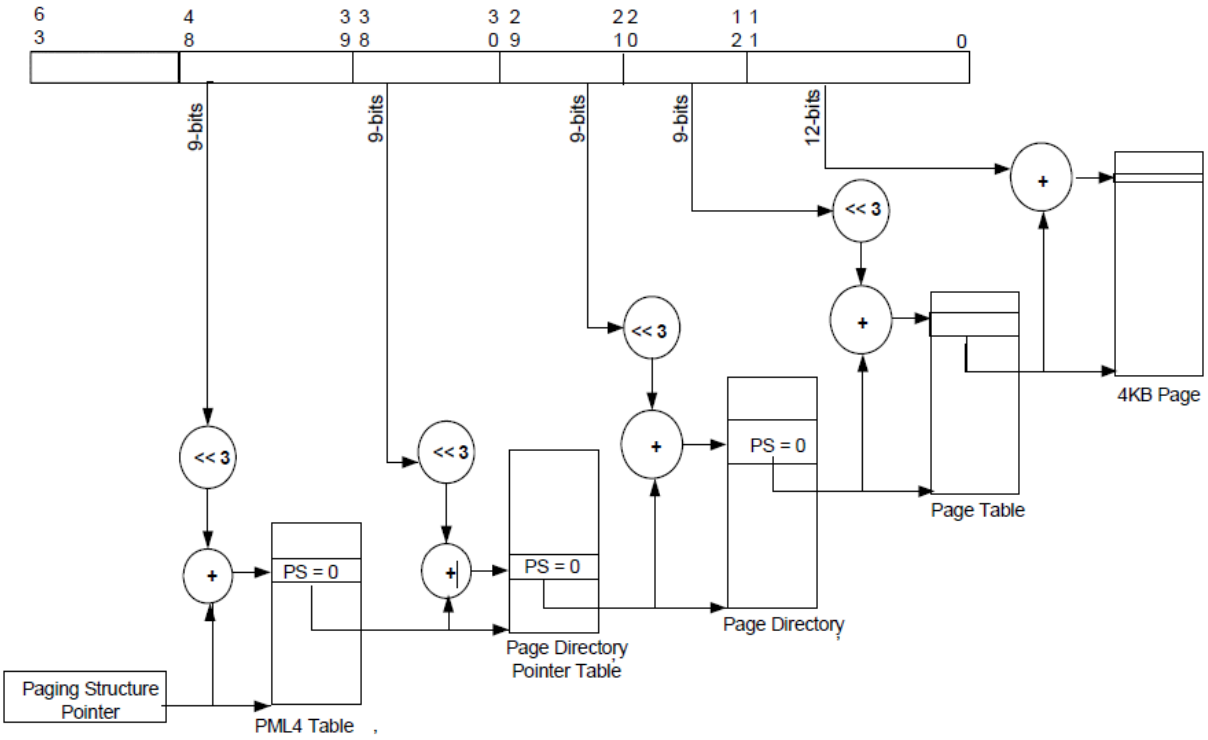



**Figure 3-9. Address Translation to a 4-KByte Page**

| 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[3] | 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 9-39. Format for Second-Level Paging Entries**

If the VT-d register Root Table Address Register has the extended bit set, the table is an instance of an **extended-root-table**, which points to **extended-context-table**. Each **extended-context-entry** contains **PASID-table** (Process Address Space ID) or **second level page table** (without PASID). The PASID-table has **PASID-entry**, which points to **first level page table**. The first level page table uses same paging structure as Intel® 64 processors in 64-bit mode.

Devices report support for requests-with-PASID through the PCI-Express PASID Capability structure. PASID Capability allows 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.

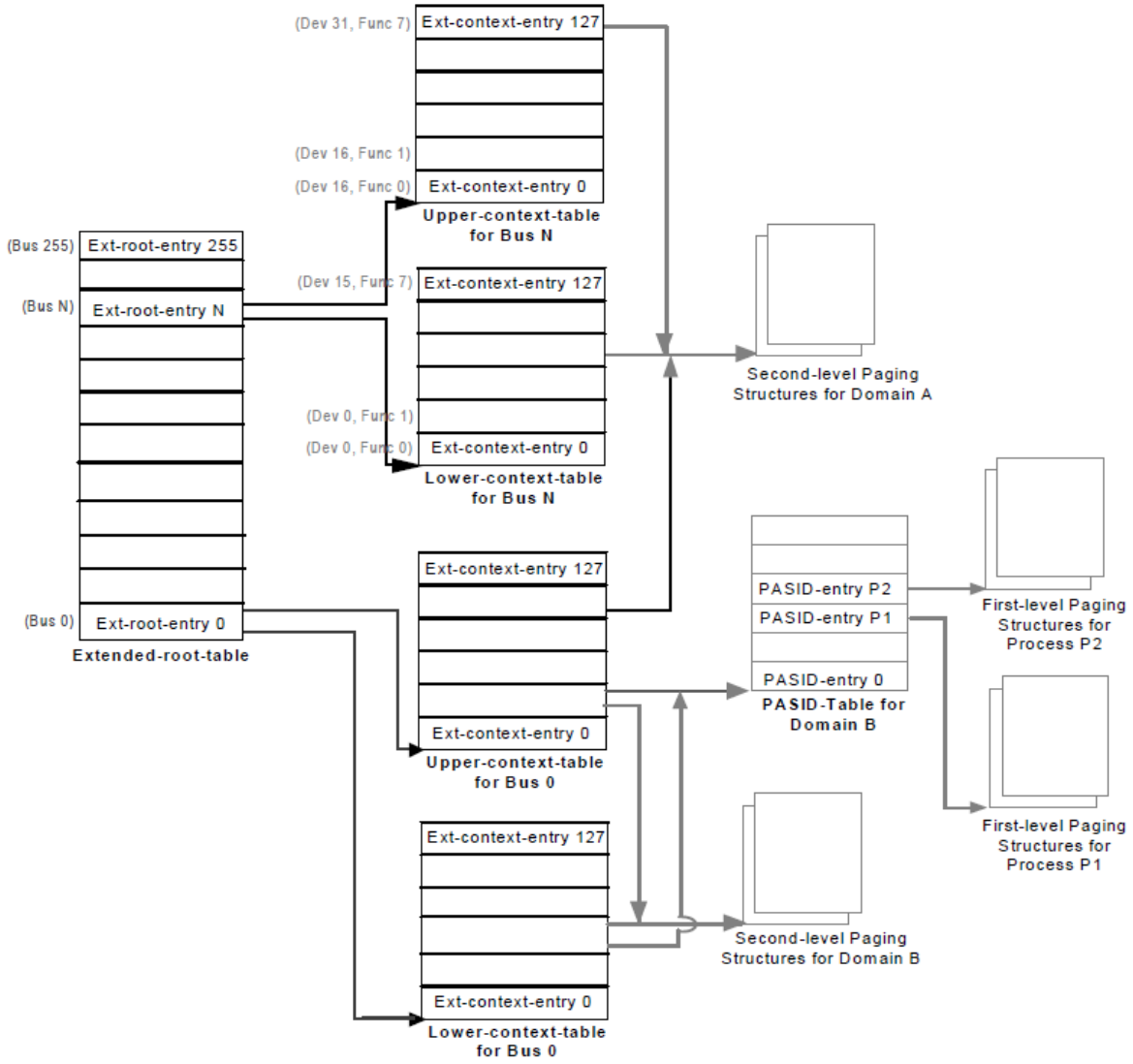



**Figure 3-8. Device to Domain Mapping Structures using Extended-Root-Table**

## BIOS responsibility

The system BIOS is responsible for detecting the remapping hardware functions in the platform and subsequently reporting the remapping hardware units to system software through the DMA Remapping Reporting (DMAR) ACPI table. For details on the ACPI table definition, please refer to the [VT-d] specification.

In the ACPI table, the first important data structure is DMA Remapping Hardware Unit Definition (DRHD). Every DMA remapping unit should have a DRHD structure associated with it. System should have at least one DRHD. Each DRHD may cover 1 or more PCI devices. For example, see below figure 8-31 from [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.

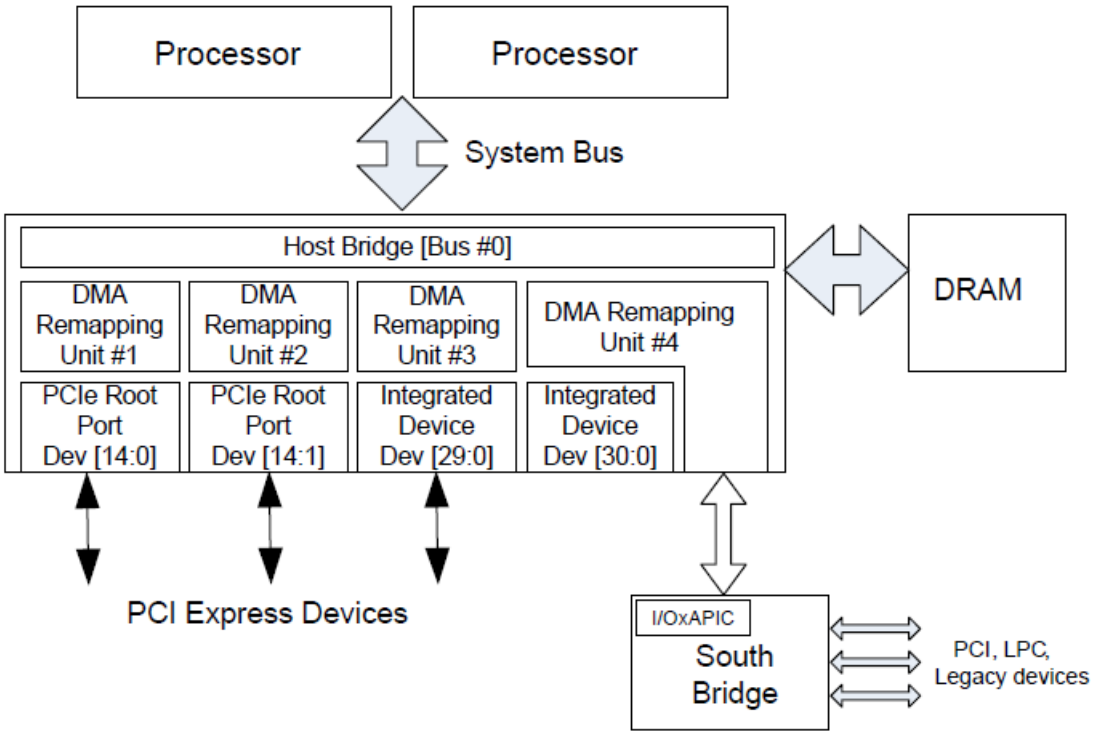



**Figure 8-31. Hypothetical Platform Configuration**

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

The second important data structure is Reserved Memory Region Reporting (RMRR). 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 that are either not DMA targets, or memory ranges that may be target of 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.

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

**Summary**
This section gives brief introduction on how Intel VT-d works.

# UEFI support for DMA

## PCI Bus for DMA

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

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

In EDKII, the PCI bus driver can be found at https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Bus/Pci/PciBusDxe. The EHCI driver is at https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Bus/Pci/EhciDxe. The XHCI driver is at https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Bus/Pci/XhciDxe. ATA AHCI driver is at https://svn.code.sf.net/p/edk2/code/trunk/edk2/MdeModulePkg/Bus/Ata/AtaAtapiPassThru.

The implementation of the PCI_IO.Map/Unmap() service forwards requests to the PCI_ROOT_BRIDGE_IO.Map/Unmap() services directly. The PCI Root Bridge driver is platform specific. 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 cache are consistent. An I/O agent can perform direct memory access (DMA) to write-back memory and the cache protocol maintains cache coherency.
In EDKII, the sample PCAT PCI Root Bridge driver can be found at https://svn.code.sf.net/p/edk2/code/trunk/edk2/PcAtChipsetPkg/PciHostBridgeDxe

For some hardware where device memory and system memory is cache-coherent, such as x86, omission of this call would not be detected. But omission of this call can pose a compatibility problem, such as found when some of the above drivers were ported to ARM-based platforms and these calls were not appropriately used.

## Support Greater-Than 4GiB DMA request

In some platforms, the DMA operation is only supported for 32-bit memory addresses, or for less-than-4GiB. But the requested host memory might be greater than 4GiB. For BusMasterCommonBuffer, this scenario should not happen because the BusMasterCommonBuffer will be allocated by PCI_IO.AllocateBuffer, which guarantees the buffer is suitable for DMA. For BusMasterRead or BusMasterWrite, in the Map() function, the PCI Root Bridge may allocate below 4GiB memory, and return this sub-4GiB memory. Then it

synchronizes greater-than-4GiB memory content to below-4GiB memory before BusMasterRead in Map(), or it synchronizes below-4GiB memory content to greater-than-4GiB memory after BusMasterWrite in Unmap(). Finally the allocated below-4GiB memory will be freed in the Unmap() function.

**Summary**

This section describes UEFI support for DMA and the EDKII implementation of a PCI driver to support DMA requests.

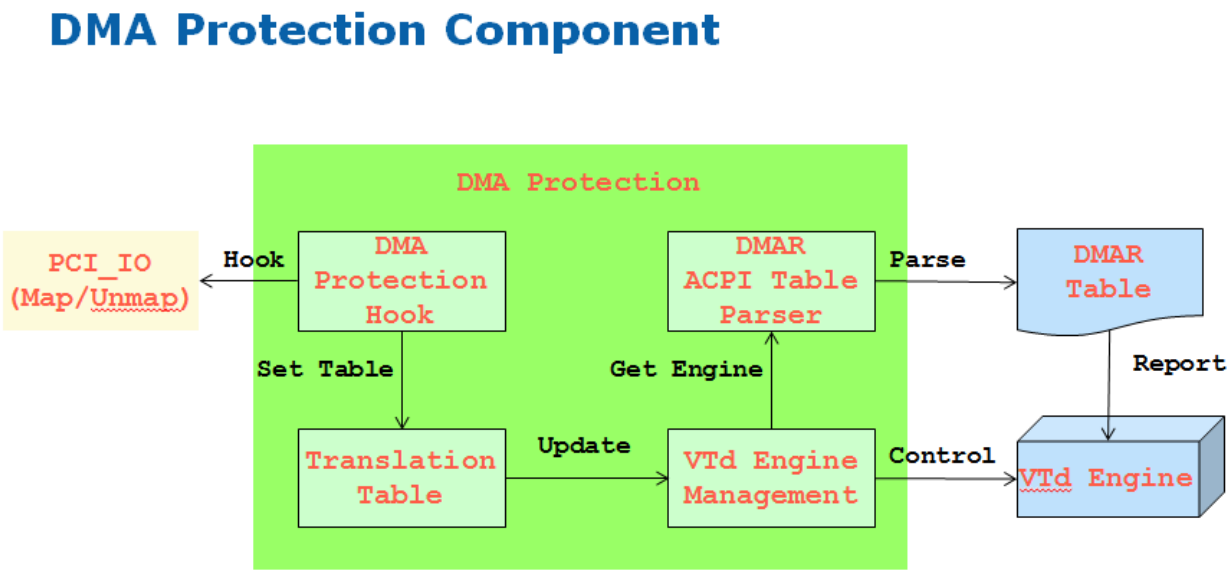
# Using VT-d in UEFI BIOS

Currently, most UEFI BIOS implementations only expose the DMAR ACPI table to report the VT-d capability. These implementations do not setup device translation table or enable the DMA remapping unit. Now, with more and more attacks to boot loader and firmware, there is a requirement to let firmware enable DMA just for the boot device and protect the physical memory from unauthorized internal DMA and all external DMA. As such, configuring the DMA remapping unit and use it in firmware can meet such requirement.

This problem is aggravated by BIOS extensions, or UEFI drivers and applications. Even though art like UEFI Secure Boot [SB] can identify the provenance of code, this identification does not necessarily prevent malicious behavior by these extensions. In addition, some platforms may not enable UEFI Secure Boot or operate with the feature disabled by the platform owner.

In rest part of this section, we will introduce how to use DMA remapping unit in firmware. We assume this work is done by a driver named the DMA protection driver.

### DMA protection driver component




The DMA protection driver consists of 4 major components:

1) DMAR ACPI table parser: It parses DAMR table and retrieves the DRHD on DMA remapping unit information and RMRR on reserved memory information.
2) VT-d engine management: It accesses the DMA remapping unit hardware register to enable or disable DMA remapping.
3) Translation table: It sets up DMAR root table, context table, PASID table, first level page table, or second level page table for PCI devices.

4) DMA protection hook: It hooks the PCI_IO.Map/Unmap function. Once the PCI device driver requests DMA access via PCI_IO.Map, the DMA protection driver can grant DMA access in translation table. After the DMA transaction is finished, the PCI device driver may call PCI_IO.Unmap to free resource, at which point the DMA protection driver needs to revoke DMA access in the translation table.

## Step1: Scan PCI hierarchy

In order to set up the DMA remapping unit, the DMA protection driver needs to understand the entire PCI hierarchy. In EDKII, the PCI bus driver will perform PCI bus enumeration and assign memory-mapped I/O (MMIO) and IO resource for each of the PCI devices. After that, the PCI bus driver will install the PCI enumeration complete protocol, and then install a PCI_IO protocol instance for each PCI devices. The DMA protection driver can register for PCI_IO protocol notification so that once a new PCI_IO is installed, the DMA protection driver knows the PCI information and records it to a global variable inside the DMA protection driver.

## Step2: Parse DMAR ACPI table

The UEFI BIOS will report DMA remapping capability via the DMAR ACPI table. In most platforms, this DMAR table is installed by a silicon driver, and it will be ready before END_OF_DXE event, which is defined in PI specification. As such, the DMA protection driver registers for END_OF_DXE event notification. Once the platform signals this event, the DMA protection can get the ACPI table from the UEFI system configuration table, which is defined in UEFI specification.

The DMA protection driver needs get the DRHD on DMA remapping unit information and save the content in a driver global variable. The DMA protection driver also needs to get RMRR on system reserved memory information and always grant DMA access to this memory. In current BIOS implementations, most platforms will report the reserved memory for legacy USB, and video buffer for legacy video option ROM of the integrated graphic device. After this, the VMM/OS will setup 1:1 mapping DMA translation table for them during system boot to make sure those devices still working.

## Step3: Setup DMAR translation table

Once the DMA protection driver gets the PCI hierarchy and DMA remapping unit information, it can set up the translation table.

The root-table and context-table should only set valid entries for the PCI devices declared in current DRHD. For those PCI devices not declared in DRHD, the root-table entry or context-table entry should be all zero. In order to save space, all context-table entries can point to one translation table entry per the DMA remapping unit. Different DMA remapping units should have different translation tables because the DMA remapping unit may have different capabilities, such as 1GiB/2MiB page support or Guest Address Width.

By default the translation table entry should have a 1:1 mapping entry but with no access rights on execute/write/read. Write/Read access rights for certain memory can be granted to certain devices when the service PCI_IO.Map receives a request from a PCI device driver. Write/Read access rights for certain memory must be revoked from certain device when PCI_IO.Unmap is invoked from a PCI device driver.

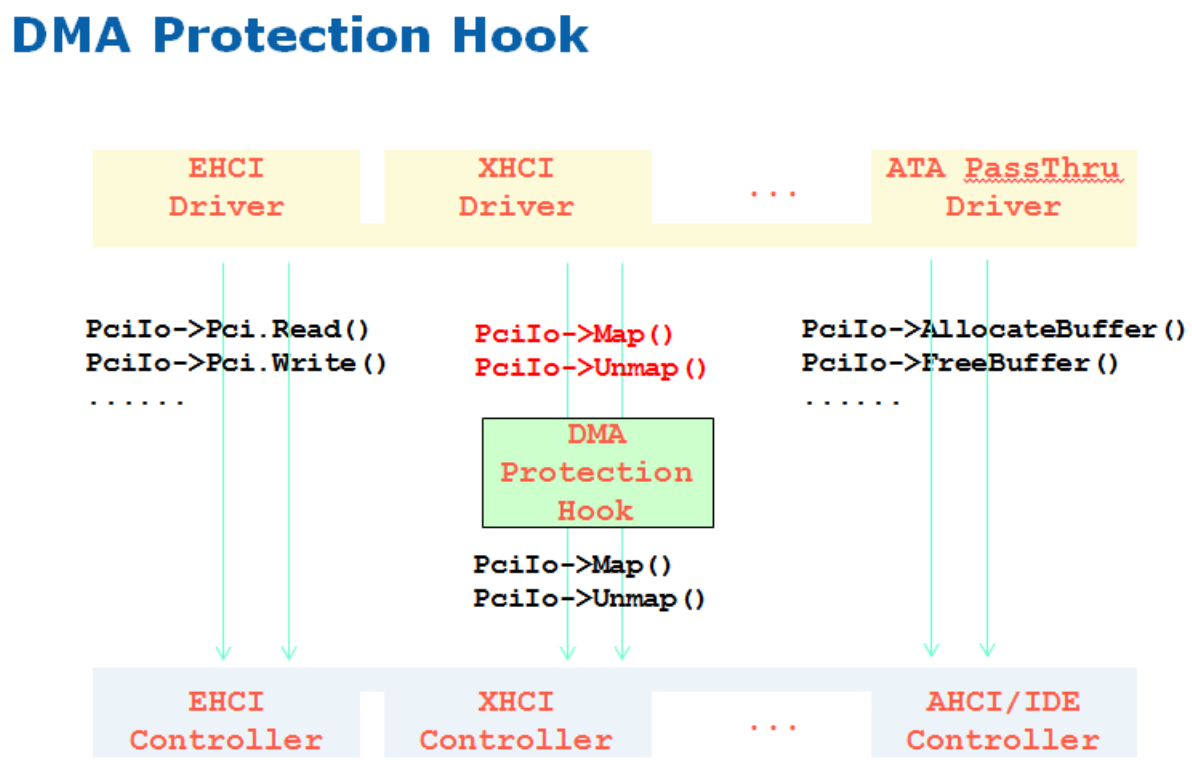
### Step4: Hook PCI_IO protocol Map/Unmap

The DMA protection driver needs to know when it should grant DMA access right to certain devices. In order to have such knowledge, the driver must know when PCI_IO.Map/Unmap is invoked. There are at least 2 options here, including:

1) Add a hook for the PCI_IO protocol Map/Unmap function.
2) Update the PCI Root Bridge IO driver to integrate DMA protection capability.

In this study we review a DMA protection driver as a stand-alone solution with could be built into any UEFI-conformant implementation on Intel hardware, we only discuss option 1. All the techniques covered can be adopted in option 2, too.

The DMA protection driver needs to record the original PciIo->Map/Unmap function pointer and replace it with Map/Unmap function in DMA protection driver. A system may have multiple PCI_IO protocols, so every PCI_IO protocol should be hooked.



## Step5: Enable DMA remapping

After the DMAR translation table is set up properly, DMA remapping can be enabled. Each DMA remapping unit has a set of standard remapping hardware registers. The DMA protection driver can set the TE (translation enable) bit in Global Command Register to enable DMA remapping.

After that, the DMA access to system memory is not granted by default, and only DMA access to the memory declared in RMRR will be granted.

## StepX: Grand/Revoke DMA access right when UEFI BIOS running

Once the DMA protection driver has a way to know the PCI_IO.Map/Unmap service locations, it can update the translation table according to the Map/Unmap request. If a PCI device driver calls Map with HostAddress and NumberOfBytes, the DMA protection driver needs to find the translation table for this PCI device, update translation table to grant access right for the requested memory, and flush Translation Lookaside Buffer (TLB) so that the new entry can take effect. The DMA protection driver needs to increase the count of access and record this value in the hardware-ignored field of the translation table because we need consider the situation that multiple Map functions request access to the same memory.

If a PCI device driver calls Unmap, the DMA remapping unit needs to find the corresponding mapping, and then find the translation table for this PCI device. The DMA remapping unit needs to decrease the count of accesses and revoke access rights if and only if the count of accesses becomes zero. This is important because the DMA remapping unit need to make sure the access is revoked in last Unmap operation.

## StepZ: Disable DMA remapping

The DMA remapping unit may not be able to remain enabled during OS boot. When OS takes control, the OS may access hardware directly to setup DMA. The PCI_IO.Map hook takes no effect in this case since all of the drivers listed are boot-services only and dematerialize upon invocation of the ExitBootServices() call from the OS loader. As such, we have to disable DMA remapping as the last OS boot step. UEFI specification defines EXIT_BOOT_SERVICES event, which is last action in BIOS and the OS loader will signal this event to tell the UEFI BIOS that it is time to teardown services. The DMA protection driver can register this event and disable DMA remapping in this event callback function. Then the OS can parse the DMAR table, set DMA translation table according to OS, and enable DMA remapping again to protect OS kernel data structure.

**Summary**
This section describes the details on how to use VT-d in a UEFI BIOS.

# Known limitations and solutions

## DMA Memory Alignment

There is a basic alignment requirement in DMA remapping unit, namely 4KiB, because the unit of translation table is 4KiB. However, when the PCI device driver submits a request via PCI_IO.Map, it does not consider the 4KiB alignment. We have several options to resolve the problem, including:

1) Update all PCI device drivers to make sure each submits 4KiB-aligned requests.
2) Add such a capability in the DMA protection hook. The DMA protection hook can do similar thing as in the greater-than 4GiB DMA support. If the requested memory is NOT aligned, the DMA protection hook will allocate aligned memory and return to the PCI device driver. Then the DMA protection driver will synchronize unaligned memory content to aligned memory before BusMasterRead in Map(), or synchronize aligned memory content to unaligned memory after the BusMasterWrite in the Unmap() invocation. Finally, the allocated aligned memory will be freed in the Unmap() function.
3) Grant DMA access rights for the full page, which covers the requested memory.

Option 1) is a clean solution, but it requires many driver updates. In the current EDKII implementation, many PCI device drivers just ignore the 4KiB alignment and submit unaligned request.
Option 2) is a clean solution for the DMA protection driver, but it might have performance impacts for allocation/free, especially for large memory on disk read/write operations.
Option 3) is easiest one, but it brings the risk of granting DMA access rights to other memory. In these other memory regions there might be some important data structures.

## CSM support

The Compatibility module support (CSM) is designed to let a UEFI BIOS support a legacy OS boot. There are some special memory usages in the legacy region. For example, some PCI option ROMs use the Embedded BIOS Data Area (EBDA), some option ROMs use less-than-1MiB Low memory managed by the Post Memory Manager (PMM), some option ROM use less-than-16MiB High PMM, and some option ROMs just look for zeroed memory between 0x60000p ~ 0x88000p and use such memory directly.
The DMA protection driver does not have such knowledge regarding which legacy memory region is used by a PCI option ROM. The best way to keep such compatibility is to grant access to less-than-16MiB memory directly and make sure there are no important data structures there.

## Driver does not follow UEFI specification

Last but not least, the DMA protection hook relies upon the PCI device driver implementation to follow the UEFI specification dictum to call Map/Unmap. If a device driver fails to follow this rule and uses memory directly, the DMA access is not granted and the device driver will get an error from the respective device hardware register. There is no way to resolve this flaw but fix the bug in such a device driver.

**Summary**
This section describes the known limitations and possible solutions to using VT-d in practice.

# Conclusion

VT-d is attractive feature for IO virtualization. It can also be used for DMA protection to mitigate DMA attacks. This paper describes a way to enable the DMA remapping unit in a UEFI BIOS to meet emerging OS requirements and threats, namely, protection from internal and external DMA in the pre-boot phase.

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

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

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.

RMRR – 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.

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

# References


[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 http://esec-lab.sogeti.com/dotclear/public/publications/10-hitbamsterdam-dmaattacks.pdf

[DMA4] Blocking the SBP-2 driver and Thunderbolt controllers to reduce 1394 DMA and Thunderbolt DMA threats to BitLocker http://support.microsoft.com/kb/2516445

[EDK2] UEFI Developer Kit www.tianocore.org

[HSTI] Hardware Security Testability Specification http://msdn.microsoft.com/en-us/library/windows/hardware/dn879006.aspx

[PCIExpress] PCI Express Base Specification, Revision 3.1 http://www.pcisig.com

[SB] Nystrom, Nicoles, Zimmer, "UEFI Networking and Pre-OS Security," Intel Technology Journal, Volume 15, Issue 1, October 2011 http://www.intel.com/content/www/us/en/research/intel-technology-journal/2011-volume-15-issue-01-intel-technology-journal.html

[UEFI] Unified Extensible Firmware Interface (UEFI) Specification, Version 2.4.b www.uefi.org

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

[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) Specifications, volumes 1-5, Version 1.3 www.uefi.org

[VT-d] Intel Virtualization Technology for Directed I/O specification, Rev 2.3 http://www.intel.com/content/www/us/en/intelligent-systems/intel-technology/vt-directed-io-spec.html

[WHCK System] Windows Hardware Certification Requirements for Client and Server Systems http://msdn.microsoft.com/en-us/library/windows/hardware/jj128256.aspx

## Authors

**Jiewen Yao** (jiewen.yao@intel.com) is an EDKII BIOS architect, EDKII TPM2 module maintainer, ACPI/S3 module maintainer, and FSP package owner with the Software and Services Group at Intel Corporation.

**Vincent J. Zimmer** (vincent.zimmer@intel.com) is a Senior Principal Engineer and chairs the UEFI networking and security sub-team with the Software and Services Group at Intel Corporation.