VMware, Inc.
3401 Hillview Ave, Palo Alto, CA 94304, USA, www.vmware.com

TOE Design (ADV_TDS.3): VMM Subsystem VMware ESXi 8.0

Author:	VMware
Version:	1.0
Date:	2022-08-10
Cert-ID:	
Company:	VMware, Inc.

Version 1.0

VMware, Inc.
3401 Hillview Ave
Palo Alto, CA 94304
United States of America

http://www.vmware.com

Revision History

Version	Description of changes	Modified by	Date
1.0	Initial Draft Version	Nicholas Leuci	2022-08-10

Table of Contents

Contents

1 Introduction

This document contains a description of the TOE Design, which is required by ADV_TDS.3. Thereby the TOE is subdivided in terms of subsystems and modules,

2 Subsystems of the TOE

Subsystem: High-Level Description of the different parts oft he TOE. It needs to be described what the main purpose of the subsystem and how.

Module: Additional subdivision of the subsystems and a More detailed description about their Implementation (e.g., based on libraries)

For each sub system (all are at high level):

- High level Subsystem description
- Near source code level description
    - Need to write SFRs security enforcing, supporting and non-interfering
    - Not in near source code level detail for parts of subsystem that are related to non-interfering SFRs
    - Modules that are non-interfering, don't need to be described in near-source code level detail.

The TOE can be subdivided into the following subsystems:

- Subsystem AAA
  Here a short description about the purpose of the subsystem should be entered.

- Subsystem BBB
  Here a short description about the purpose of the subsystem should be entered.

- Subsystem CCC
  Here a short description about the purpose of the subsystem should be entered.

- …

The figure below gives an overview about the architecture of the TOE and how the TOE can be subdivided into Subsystems and modules

Put the detailed figures of the overall TOE. Put the low-level-ESXI architecture document. How we draw the boundaries of modules in the sub-system is up to us. Multiple images will be supplied.

2.1 Interaction between sub-systems

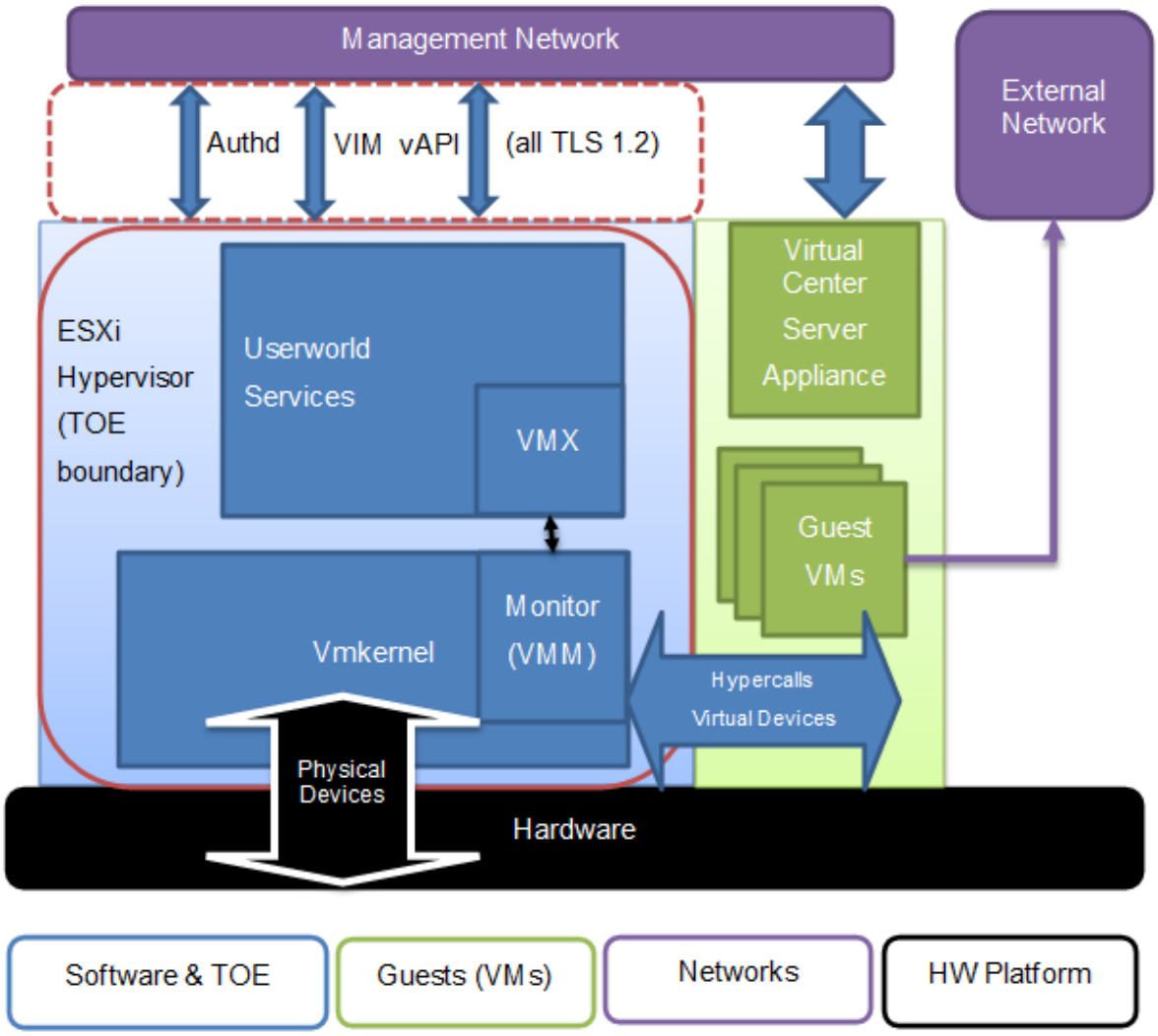


Figure 1: TOE Subsystems and Modules

.

Please give a short explanation of Figure 1. Thereby especially the purpose and the interactions between the subsystems should be explained.

2.2 Subsystem AAA

Detailed Description of the purpose and content of subsystem AAA. Each group adds their own subsystem section.

Still high-level description – half a page to one-to-two page level. Overview of which modules are in the subsystem; how the modules relate to each other. Overview of APIs of modules, i.e., communication between VMkernel and Monitor.

2.3 Subsystem BBB.

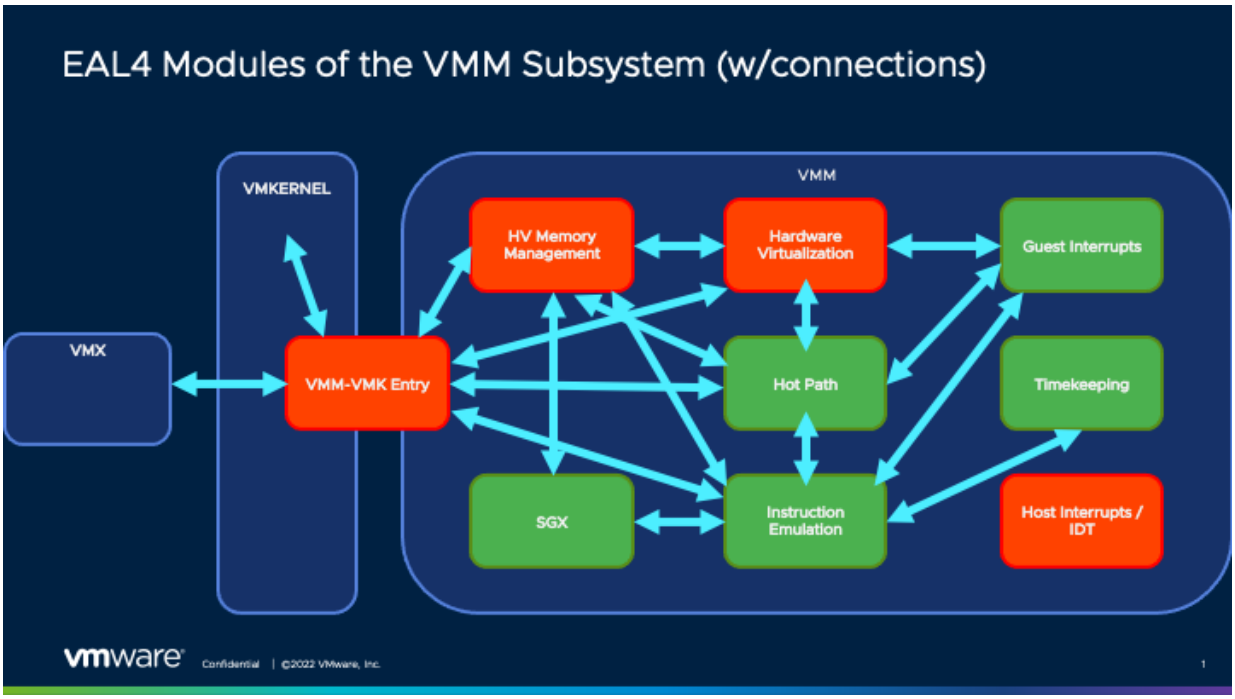
Detailed Description of the purpose and content of subsystem BBB. Each group adds their own subsystem section.

2.4 Subsystem CCC

Detailed Description of the purpose and content of subsystem CCC. Each group adds their own subsystem section.

4 Virtual Machine Monitor (VMM) Subsystem

A. Subsystem Diagram



SFR Color Encoding Legend: **Red – Enforcing modules** **Yellow – Supporting modules** **Green – Non-Interfering modules**

B. High Level Summary

The Virtual Machine Monitor (hereafter "VMM") is a kernel-mode program responsible for execution of virtual CPUs. One VMM program instance exists per VM. One VMM world (thread) exists for each virtual CPU in a VM.

VMM presents virtual hardware to the virtual machine and causes its virtual CPUs to make progress in execution, in a high-performance manner, with proper isolation and security. VMM relies on hardware virtualization (via Intel's VT) and, to a lesser extent, instruction emulation for this purpose.

VMM exposes virtual hardware to VM software and handles the edges of this interaction, including virtual interrupts and virtualized device access. VMM exposes memory to a VM as well. As such, VMM is responsible for managing views of memory.

VMM also implements and supports various virtualization features. Some features include nested virtualization (such that a VM can, internally, run a nested VM) and virtualization of CPU features such as secure enclave execution via Intel's SGX.

VMM interacts with other software in the TOE by switching to the vmKernel when required. VMM cooperates with the vmKernel (vmKernel RM CPU Subsystem) to share the host CPU

(as both pieces of software are kernel-mode, privileged software). VMM is largely subordinate to the vmKernel, as the vmKernel RM CPU Subsystem decides scheduling of host worlds such as those worlds running VMM.

C. List of Modules

Module Name	Brief Description	Security Type
Hardware Virtualization	Support for Intel's VT CPU virtualization support, both for regular VMs and those containing nested VMs. Manages and executes context-switching between VMM software in the TOE and the Virtual Machine domain (VM).	SFR-ENFORCING
HV Memory Management	Virtualization of guest memory including presentation of vmKernel-provided memory to the VM and any nested VMs run inside.	SFR-ENFORCING
VMM/VMK Entry	Context-switching between VMM and the vmKernel and associated optimizations.	SFR-ENFORCING
Host Interrupt/IDT	Optimized Inter-Processor Interrupt support for fast synchronous signaling between VCPUs of a single VM.	SFR-ENFORCING
SGX	Virtualization of Intel Software Guard Extensions support for secure enclaves.	SFR-NON-INTERFERING
Guest Interrupts	VM-internal virtual interrupt support, including delivery of virtual interrupts to virtual VCPUs for consumption in the VM.	SFR-NON-INTERFERING
Instruction Emulation	Correct emulation of instructions, as a fallback when fast handling of a VT exit is impossible, or emulation is otherwise required.	SFR-NON-INTERFERING
Hot Path	Fast handling of VT exits from the Hardware Virtualization module.	SFR-NON-INTERFERING
Timekeeping	Management of VM-perceived time.	SFR-NON-INTERFERING

4.1 VMM Hardware Virtualization (SFR-ENFORCING)

The VMM Hardware Virtualization (hereafter "HV") module runs as part of the Virtual Machine Monitor (part of the TOE, a kernel-mode program with one instance per VM, hereafter "VMM"). The HV module implements execution of the virtual machine domain (guest OS, hereafter "VM" or "guest") by use of Hardware Virtualization ("HV") provided by the Intel CPU Virtualization Technology (known by Intel as "VMX" but hereafter referred to by the VMware term, "VT"). The HV module manages CPU and VT state, and handles switches to and from VM execution, for various reasons.

Once initialized, the HV module runs in a loop:

1. enter VM execution (hereafter known as an "HV resume"),

2. wait for the CPU to exit VM execution and return to VMM (hereafter known as an "HV exit"),
3. determine from exit description what handler to run,
4. call that handler,
5. and likely return to the first step: HV resume.

The HV module is responsible for isolating the VM from the VMM (and other host software). This isolation is implemented two ways: (1) by careful constraint of and description of VM execution using VT state and (2) by careful context-switching of CPU state, avoiding undesirable effects upon VMM (and other host software).

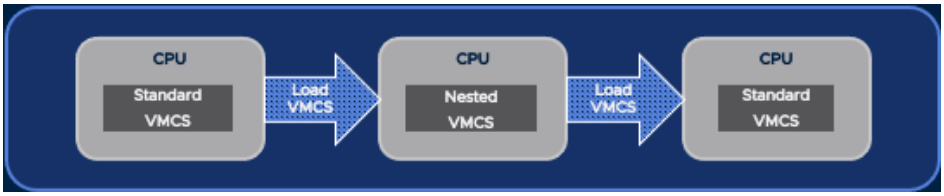
VT contains a Virtual Machine Control Structure (hereafter "VMCS") which defines state for the CPU to load upon HV resume and HV exit, as well as controls constraining VM execution and under what conditions HV exits shall occur. The HV module programs the VMCS accordingly. Intel describes the VMCS, VT and related information in Intel Software Developers Manual, Volume 3C: System Programming Guide, Part 3, Chapters 23-27 and 30 (See 4.1,7 Appendix A below).

The HV module is responsible for context-switching of VM state. This context-switching occurs in different ways, at different points in code: at HV resume and HV exit (automatically via VT), in code paths immediately before HV resume and after HV exit (in software, via the HV module), and in deferred code paths transitioning between pieces of software in the TOE (in the VMM/VMK Entry Module, 4.8).

Because the VMCS defines when the VM may cause HV exits, and because the HV module must context-switch VM CPU state (which could, unswitched, affect other software in the TOE), the HV module is SFR-enforcing for FPT_VIV_EXT.1.1. Because the handling of HV exits (which could, handled incorrectly, affect other software in the TOE) is implemented in the HV module, it is SFR-enforcing for FPT_VIV_EXT.1.2.

The HV module implements nested virtualization support, allowing a VM run encapsulated VMs of its own. The module implements Virtualized Hardware Virtualization (hereafter "VHV") and specifically for Intel, it implements Virtual VT (hereafter "VVT"). VHV is supported for Microsoft Windows guests using Hyper-V, which rely upon an implementation of VT (here, our VVT) for the Microsoft implementation of Virtualization-Based Security.

When VVT is in use, the guest software is split conceptually into two parts: the inner hypervisor (which programs and uses VVT via VT semantics) and the inner guest (which runs under control of the HV module, with additional description and constraints added by the inner hypervisor). For performance, the HV module uses two VMCS structures when running with VVT: a standard VMCS and a nested VMCS. The standard VMCS describes and constrains the inner hypervisor, while the nested VMCS describes and constrains the inner guest. Only one VMCS is active at a given time, with transitions and management of VMCS state optimized carefully.



When a VMCS is not in active use, its values may be modified by live execution (e.g. the inner hypervisor writing new configuration to the nested VMCS to constrain the inner guest).

The HV module maintains an in-memory cache of nested VMCS state and tracks dirty subportions of this cache, deferring recomposition of the nested VMCS until just before its live use.

When switching between executing the inner hypervisor and the inner guest, HV controls must be updated in the VMCS that is about to become active. This update carefully composes as safe VMCS. When transitioning to executing the inner guest, the VMCS combines the wishes of the inner hypervisor (as described in the nested VMCS) and the requirements of the HV module (as described in the standard VMCS).

For performance reasons, some VMCS fields allow the guest (be it the inner hypervisor or the inner guest) to access certain CPU resources directly, without exiting to the HV module. Such resources are those which are either inherently harmless to the TOE (e.g. guest general-purpose registers) or those which are context-switched carefully shortly after HV exit (e.g. side-channel mitigation model-specific registers which do no harm to execution of the TOE during the brief moment between HV exit and context-switching).

VT's VMCS contains controls and structures related to guest-visible memory. That state and its management is handled by The VMM HV Memory Module (4.2).

HV exit handling in the HV module will call into other modules, depending upon the exit condition.

The HV module cannot be disabled and is always used in the running of VMs.

4.1.1 Security Functionality (SF)

See the table in Section 4.1.2.

4.1.2 Security Functional Requirement (SFR)

Security Function (SF)	Security Function Requirement (SFR)	Rationale
SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	The TSF shall maintain a security domain for the execution of each virtual machine that protects the virtual machine from interference and tampering by untrusted subjects or subjects from outside the scope of the VM.
SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.2	The TSF shall enforce separation between the security domains of VMs in the TSC.

4.1.3. Provided TSFI

This module has no TSFI as it is an internal module and has no exposure to outside the TOE.

4.1.4.1 Internal Interfaces (Context-switching between VMM and VM).

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
HVResume	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None**. Does not return to caller. Exits at VMCS HOST_RIP.	Main entry point to HV resume functionality in VMM.
HV_StepToSafePointAndResume	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None**. Does not return to caller. Exits at VMCS HOST_RIP.	Alternate path to call VMM Instruction Emulation Module (see 4.5), then program VMCS with VMM state and proceed to resume.
HVVendorSpecificResume	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None**. Does not return to caller. Exits at VMCS HOST_RIP.	Program VMCS with VM state (program counter, stack pointer, CPU flags, pending interrupt information if any)
HVMSR_VMEnter	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None**. (void function).	Reload any software-switched Model-Specific Registers to VM values.
HVResumeLowLevel	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None**. (Assembly function taking no arguments). Does not return to caller.	Load VM general-purpose register state into CPU, execute VT "vmresume" instruction to actuate switch from VMM to VM

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
HVExitLowLevel	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1, FPT_VIV_EXT.1.2	**None**	**None**. (Assembly function entered directly by hardware). Does not return to caller as there is no caller.	Main exit point from VM back to VMM. Saves VM general-purpose register state from CPU and immediately loads zeroes into most such registers. VMM software is now in control of the CPU.
HVVendorSpecificExit	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1,	**None**	**None**. Does not return to caller.	Saves more VM state, reloads more VMM state.
HVMSR_VMExit	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None**. (void function).	Reloads any software-switched Model-Specific Registers back to VMM values.
HVExit	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**reason** - VT exit reason, **idtVecInfo** - VT-provided IDT vectoring information	**None** (void function), does not return.	Processes exit cause provided by VT, calling various other code to handle each type of exit. (TBD: explain more/better) .

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
VVT_VMENTER	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**launch** - whether launching (the first entry, before having resumed) or resuming **instrLen** - the length of the instruction attempting the launch (to advance past, upon success)	**An x86fault object pointer**, representing either a specific failure, or successful VM entry (X86Fault_None or similar).	Switch to the nested guest VMCS, ready for HV resume into the nested guest, if successful.

4.1.4.2 Internal Interfaces (Sensitive host fields for context-switch)

Module Function	Security Function(s)	SFR(s)	VMCS fields protected	Parameters	Return Value	Rationale
HVVTInitVMCSHostFields	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	HOST_CS, HOST_ES, HOST_SS, HOST_DS, HOST_TR, HOST_PAT, HOST_EFER, HOST_CR3, HOST_TRBASE, HOST_GDTRBASE, HOST_IDTRBASE, HOST_RSP, HOST_RIP	**None**	**None**. (void function).	Basic host register state loaded upon HV exit. Static after initialization. Provides program counter (HOST_RIP) to execute HVExitLowLevel, stack pointer (HOST_RSP) and other basic state, automatically switched by VT support in the CPU. Ensures fundamental register state in VMM unaffected by VM values.
HV_SetHostCR0	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	HOST_CR0	**None**	**hostCR0** - value to set in cr0	Another fundamental control register loaded upon HV exit.
HV_SetHostCR4	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	HOST_CR4	**None**	**hostCR4** - value to set in cr4	Another fundamental control register loaded upon HV exit.

Module Function	Security Function(s)	SFR(s)	VMCS fields protected	Parameters	Return Value	Rationale
HV_SetNestedPagingRoot	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	EPTP	**l4MPN** - MPN corresponding to EPTP to populate.	**None**. (void function).	Sets the VT nested paging root (VMCS field EPTP) to a given value. (See module 4.2: VMM guest memory)

4.1.4.3 Internal Interfaces (Sensitive host field for posted interrupts)

Module Function	Security Function(s)	SFR(s)	VMCS fields protected	Parameters	Return Value	Rationale
HVVTInitPostedInterrupts	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	PI_NOTIFY, PI_DESC_ADDR	**None**	**None**. (void function).	Allow VT operation to use posted interrupts without incurring an HV exit when the notification vector is used to raise an inter-processor interrupt from another CPU. If the vector were mis-programmed, interrupts could be dropped, resulting in host stability.

4.1.4.4 Internal Interfaces (Controls determining circumstances causing HV exits)

Module Function	Security Function(s)	SFR(s)	VMCS fields protected	Parameters	Return Value	Rationale
HVSetVMCSPinCtl	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	PIN_VMEXEC_CTL	**None**	**None**. (void function).	Configures handling (whether to HV exit) of asynchronous events such as interrupts (including host interrupts unrelated to the currently-running VM). Careful programming of these controls guarantees VM interruptibility and allows host software in the TOE (VMM, the vmkernel) to operate effectively.
HVSetVMCSCPUCtl	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	CPU_VMEXEC_CTL, 2ND_VMEXEC_CTL	**None**	**None**. (void function).	Configures handling (whether to HV exit) of synchronous processor events (mostly execution of specific instructions and related circumstances). Used to inhibit direct access to sensitive host resources (e.g. port I/O instructions on the physical CPU) and to otherwise constrain VM execution.
HVSetVMCSExitCtl	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	VMEXIT_CTL	**None**	**None**. (void function).	Configured automatic actions performed by the CPU at VT exit, such as entering long mode (64-bit execution, as requited by the TOE – see VT_REQUIRED_EXIT_CTLS).
HVSetVMCS2ndCtl	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	2ND_VMEXIT_CTL	**None**	**None**. (void function).	Configured automatic actions performed by the CPU at VT exit (secondary list). For example, whether EPT is enabled (used by module 4.2: VMM HV Memory Management).
HVSetVMCSXCPCtl	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	XCP_BITMAP	**None**	**None**. (void function).	Force #AC exceptions to HV exit. Without this, a CPU can be caught in an infinite #AC loop due to a malicious VM. Force machine checks to exit, to be reported to the host kernel. See also HV_XCP_MASK. Allows other forcing of exceptions to exit, as well.

Module Function	Security Function(s)	SFR(s)	VMCS fields protected	Parameters	Return Value	Rationale
HVSetMSRBitmap	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	MSRBITMAP	**None**	**None**. (void function).	Enables bitmap allowing for non-exiting access to specific Model-Specific Registers, which are in turn context-switched in HVExit/HVResume when made accessible this way.
HV_SetMSRIntercept	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	MSRBITMAP	**bitmap** - the bitmap address, **msrNum** - the MSR to intercept, **accessMode** - the read/write access to intercept	**None**. (void function).	Denies non-exiting access to a specific Model-Specific Register.
HV_ClearMSRIntercept	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	MSRBITMAP	**bitmap** - the bitmap address, **msrNum** - the MSR not to intercept, **accessMode** - the read/write access not to intercept	**None**. (void function).	Allows non-exiting access to a specific Model-Specific Register.
HVSetVMCSEnclsBitmap	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	ENCLS_EXITING_BITMAP	**None**	**None**. (void function).	Enables bitmap allowing for non-exiting execution of ENCLS instruction for some situations. See module 4.8: SGX.
HVSetVMCSEnclvBitmap	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	ENCLV_EXITING_BITMAP	**None**	**None**. (void function).	Enables bitmap allowing for non-exiting execution of ENCLV instruction for some situations. See module 4.8: SGX.

4.1.5 Used interfaces of other modules

Module Function	Module	Description	Parameters	Return Value	File
HVTryFastExit	4.4 Fast Path	Attempts to handle VT exit information quickly	**reason** - VT exit reason	**None** (void function), does not return if handled.	vmcore/monitor/vmm/hv/vt/hv-common.h
HVTryFastNestedExit	4.4 Fast Path	Attempts to handle VT exit information quickly, for inner guest execution	**reason** - VT exit reason, **idtVecInfo** - VT-provided IDT vectoring information	**None** (void function), does not return if handled.	vmcore/monitor/vmm/hv/vt/hv-common.h
Interp_Step	4.5 Instruction Emulation	Emulates one guest instruction, delivering any resulting faults to the guest. Run-time entrypoint to the interpreter.	**None**.	**None** (void function)	vmcore/monitor/common/cpu/x86/interp.c
MonMSR_SetMSR	4.8 VMM/VMK interface	Communicates to switching interface properties of the given MSR (whether it must be reloaded and with what value, when entering/exiting VMM)	**msr** - Model-Specific Register (from short list of allowed values), **newVal** - new value for MSR, **flags** - switching reload properties	**None** (void function)	vmcore/public/monMSR.h
MonMSR_SetMSRUnused	4.8 VMM/VMK interface	Communicates to the switching interface that the given MSR does not need to be reloaded when entering VMM (VMM can run with any value, without ill effect).	**msr** - Model-Specific Register (from short list of allowed values)	**None** (void function)	vmcore/public/monMSR.h

4.1.6 Mapping to the Source Code

Function	Description	File
HVResume	Entry point for HV resume flow.	vmcore/monitor/vmm/hv/common/hv.c
HV_StepToSafePointAndResume	Corner case for emulation before HV resume.	vmcore/monitor/vmm/hv/common/hv.c
HVVendorSpecificResume	VT-specific HV resume, next step after HVResume	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVMSR_VMEnter	Model-Specific Register switch	vmcore/monitor/vmm/hv/common/hvMSR.c
HVResumeLowLevel	Final switch of state and actual transition to VM execution	vmcore/monitor/vmm/hv/vt/vtasm.S
HVExitLowLevel	Initial switch from VM execution, save of VM state	vmcore/monitor/vmm/hv/vt/vtasm.S
HVVendorSpecificExit	VT-specific HV exit	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVMSR_VMExit	Model-Specific Register switch	vmcore/monitor/vmm/hv/common/hvMSR.c
HVExit	General HV exit path, calls out to various handlers for exit reasons	vmcore/monitor/vmm/hv/common/hv.c
(VMCS field table in VMW notation, no named functions)	Tokens naming VMCS fields used in VMW code and definitions via preprocessing	vmcore/public/x86vt-vmcs-fields.h
HVVTInitVMCSHostFields	Set initial/static VMCS host fields to reload from at HV exit	vmcore/monitor/vmm/hv/vt/hv-vt.c
HV_SetHostCR0	Set host VMCS %cr0 register field to reload at HV exit	vmcore/monitor/vmm/hv/vt/hv-vt.c
HV_SetHostCR4	Set host VMCS %cr4 register field to reload at HV exit	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVVTInitPostedInterrupts	Initialize posted interrupt state in VMCS, if enabled.	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVSetVMCSPinCtl	Set asynchronous event ("PIN") controls.	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVSetVMCSCPUCtl	Set synchronous CPU event controls.	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVSetVMCS2ndCtl	Set secondary synchronous CPU event controls.	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVSetVMCSExitCtl	Set VM exit behaviors.	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVSetVMCSXCPCtl	Set exception exiting controls.	vmcore/monitor/vmm/hv/vt/hv-vt.c

Function	Description	File
HVSetMSRBitmap	Set bitmap of Model-Specific Registers used to determine whether read/write MSR instructions cause VM exits or write to CPU state (requiring context-switching in software).	vmcore/monitor/vmm/hv/vt/hv-vt.c
HV_SetMSRIntercept	Set access VMM interception (force HV exit) in given MSR bitmap for given MSR and access type.	vmcore/monitor/vmm/public/hvPlatform.h
HV_ClearMSRIntercept	Clear access VMM interception (avoid HV exit) in given MSR bitmap for given MSR and access type.	vmcore/monitor/vmm/public/hvPlatform.h
HVSetVMCSEnclsBitmap	Set bitmap for ENCLS-instruction exiting (see module 4.9: SGX)	vmcore/monitor/vmm/hv/vt/hv-vt.c
HVSetVMCSEnclvBitmap	Set bitmap for ENCLV-instruction exiting (see module 4.9: SGX)	vmcore/monitor/vmm/hv/vt/hv-vt.c
HV_SetNestedPagingRoot	Sets the VT nested paging root (VMCS field EPTP) to a given value. (See module 4.2: VMM guest memory)	vmcore/monitor/vmm/hv/vt/hv-vt.c
VVT_VMENTER	Effects a VM entry managed by the current VMCS.	vmcore/monitor/common/hv/vt/vvt.c

4.1.7 Appendix A: Bibliography for the Intel VT References

Document	Author / Company	Date	Notes
Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3	Intel Corporation www.intel.com	06/30/2022	Specific citations are in the detail table below. **Note:** Intel renumbers resources over time. These chapters and the SDM volume number are correct as of June 30, 2022.
Chapter name	**Content**		
(23) Introduction to Virtual Machine Extensions	"VMX" / VT overview		
(24) Virtual Machine Control Structures	VMCS definitions		
(25) VMX Non-Root Operation	Guest operation		
(26) VM Entries	Transitions from VMM to VM		
(27) VM Exits	Transitions from VM to VMM		

(30) VMX Instruction Reference	Instructions for VMCS programming, VM resume, HV CPU state manipulation

4.1.8 Appendix B: Navigating HV module code

The code and header files implementing the HV module are used across multiple products and CPU architectures. Only a subset of the code is of relevance to the TOE. This table endeavors to simplify reading code and header files by explaining what is included and excluded from the TOE. Terminology clarifying the above documentation is also provided.

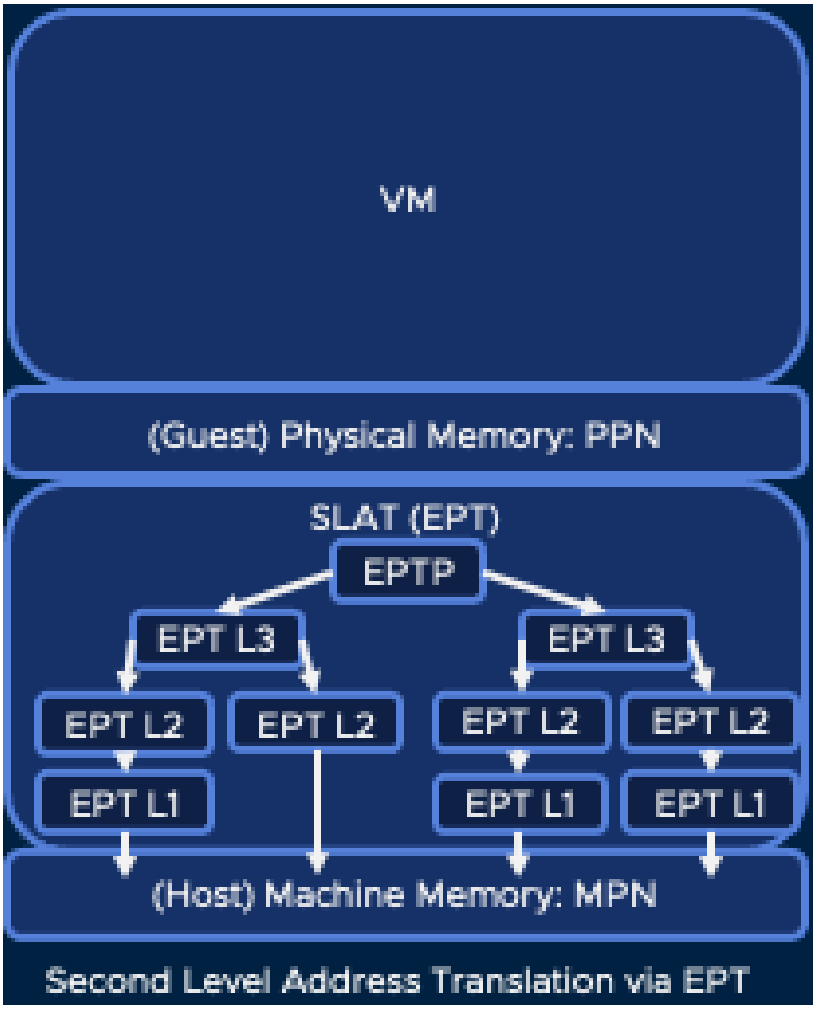
Term or token	Meaning	Included in TOE?
vmx86_server	Set to 1 if building ESX	Yes
VMX86_SERVER (CPP token)	#defined if building ESX	Yes
SERVER_ONLY()	Macro contents defined if building ESX	Yes
HOSTED_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_vmm	Set to 1 if building VMM	Yes
vmx86_ulm	Set to 0 if building VMM	No
ULM_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_release	Set to 1 if building for releases to customers	Yes
vmx86_debug	Set to 1 if building for debug builds	No
vmx86_devel	Set to 1 if building for internal developers	No
vmx86_vt	Set to 1 if building for VT support	Yes
vmx86_svm	Not relevant to VT support (AMD-specific)	No
VCPU_InGuestOperation()	Returns TRUE if the VCPU is running or emulating the nested guest	Yes

4.2 VMM HV Memory Management (SFR-ENFORCING)

The VMM HV Memory Management (hereafter "guest memory") module implements management of memory pages accessible to the virtual machine domain (guest OS, hereafter "VM") while it executes in HV (via module 4.1: VMM Hardware Virtualization) or emulation (via module 4.5: VMM Instruction Emulation).

Intel provides a technology for Second Level Address Translation ("SLAT") known as Extended Page Tables (hereafter "EPT"). EPT is a hierarchical system of translation via page tables: 4 kilobyte pages of 512 64-bit Extended Page Table Entries (hereafter "EPTEs") apiece, from a root known as the Extended Page Table Pointer (hereafter "EPTP") on to some terminal EPTE. ETPEs also encode access permissions. Intel describes EPT in the Intel Software Developers Manual, Volume 3C: System Programming Guide, Part 3, Chapter 28 (See 4.2.7 Appendix A below).

The guest memory module manages views of guest physical memory using EPT. When the VM executes via Intel's VT (see 4.1: VMM Hardware Virtualization), the guest memory module provides this view of guest physical memory for a given VCPU via an EPTP. The EPT tree translates between guest Physical Page Numbers (hereafter "PPNs") and host Machine Page Numbers (hereafter "MPNs") or non-present entries. When in VT, a memory access will obey the programmed EPT tree and result in either a successful, fast memory access or an HV exit to VMM.



Because the guest domain can directly access host memory (as provided, constrained and prescribed by the TOE), this module is SFR-enforcing. The module must provide only the correct pages of host memory, and guarantee that access is correctly constrained.

The guest memory module programs the EPT tree with pages of memory and permissions. The guest memory module interfaces with the host memory allocator (see 9.5: "VM Volatile Memory Virtualization") to acquire the correct page and any constraining page permissions.

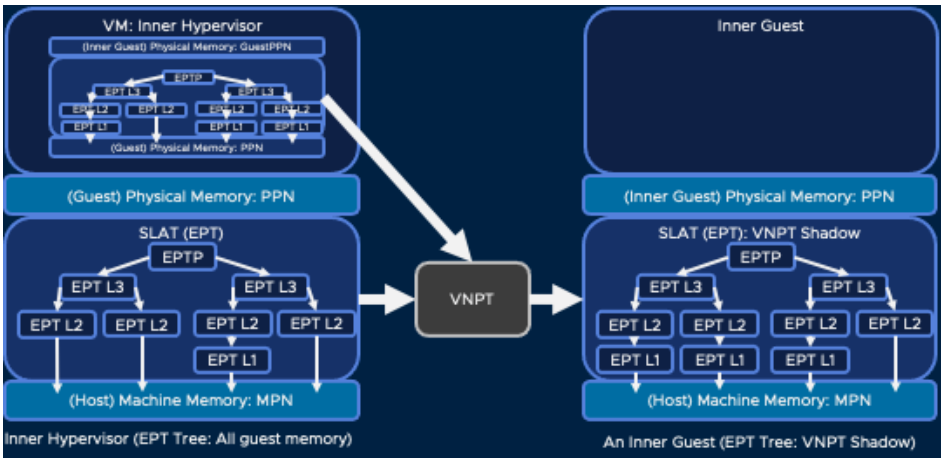
The memory allocator module may also request that the guest memory module relinquish access to a page.

The guest memory module, vmKernel memory modules and other virtualization modules use an intermediate representation of memory known as the "memory bus" (or "BusMem") to prioritize what type of resource is visible for a given PPN. It is possible, for example, to layer a virtual device (e.g. SVGA) on top of non-volatile memory, such that an access to a particular PPN should exit to VMM and be handled by SVGA device code. The BusMem system denominates pages in BusMem Page Numbers (hereafter "BPNs"). In the conversion of a PPN to an MPN, the memory bus is traversed.

For efficiency, contiguous, aligned sets of PPNs of size 512 or 512 * 512 with identical permissions may be promoted to a larger page size. Thus 512 aligned, contiguous 4 kilobyte pages mapped at level 1 of EPT may be replaced with one 2 megabyte page at level 2 of EPT, and 512 aligned, contiguous 2 megabyte pages mapped at level 2 of EPT may be replaced with one 1 gigabyte page at level 3 of EPT. If permissions on a subpage of any larger (2 megabyte, 1 gigabyte) page are then modified, the larger page is invalidated. Such optimizations add complexity to the module, but do not violate its security guarantees, as protections are always enforced conservatively and correctly for every page.

EPT uses Translation Lookaside Buffers (EPT "TLBs" hereafter) in the CPU to ensure high performance. These caches are tagged with Virtual Process Identifiers (hereafter "VPIDs"). As such, the module must follow cache coherency protocols when unmapping or modifying mappings in EPTEs. The guest memory module coordinates with the VMM-VMK Entry module (see 4.8) to ensure proper switching and flushing of EPT TLBs and VPIDs when switching between VMs.

The HV module (see 4.1: VMM Hardware Virtualization) implements virtualization of Hardware Virtualization (VHV) via virtualization of Intel's VT technology (VVT). The guest memory module provides complementary technology to virtualize guest memory via virtualization of Intel's EPT, implemented as Virtualization of Nested Page Tables (hereafter VNPT). When a VM uses VVT, its inner hypervisor describes execution of its inner guests via VT semantics. When a VM uses VVT, its inner hypervisor may use VNPT, describing inner guest memory via EPT semantics. VNPT converts inner hypervisor description of inner guest memory to a host-level EPT tree known as a VNPT shadow, for efficient execution. VNPT shadows are composed of strict subsets of a VM's primary EPT tree, with page protections at least as restrictive. Thus each VCPU can either run with the VM-global EPT tree or one of its subsets, a VNPT shadow.



The guest memory module also maintains parallel x86 page table trees used to enable fast emulation of guest memory accesses. These trees, known as the trace tree and no-trace tree, map views of guest physical memory into VMM. These trees are not SFR-enforcing as they do not directly expose memory to the guest, but they are noteworthy in support of other modules (e.g. 4.5: VMM Instruction Emulation).

4.2.1 Security Functionality (SF)

See the table in Section 4.2.2.

4.2.2 Security Functional Requirement (SFR)

Security Function (SF)	Security Function Requirement (SFR)	Rationale
SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	The TSF shall maintain a security domain for the execution of each virtual machine that protects the virtual machine from interference and tampering by untrusted subjects or subjects from outside the scope of the VM.

4.2.3 Provided TSFI

This module has no TSFI as it is an internal module and has no exposure to outside the TOE.

4.2.4.1 Internal Interfaces of the Module (General execution)

Module Function	Security Function (s)	SFR(s)	Parameters	Return Value	Rationale
GPhysSetPTE	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**dest** - pointer to PTE to set, **newVal** - value to set in PTE	**None** (void function).	Sets a PTE in a GPhysTree (EPT tree).
GPhysClearPTE	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**dest** - pointer to PTE to clear, **pa** - physical address represented by this PTE **level** - level in page table hierarchy at which this entry is wired	**GPhysPTE** - the previous value of this PTE.	Clears a PTE in a GPhysTree (EPT tree). Also clears VMM software mapping ("NPTMap") of the EPT subtree wired by the previous PTE.

Module Function	Security Function (s)	SFR(s)	Parameters	Return Value	Rationale
GPhysProtectPPN	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**bpn** - (argument not used) **ppn** - page to protect **trace** - software callback pointer to apply, if any **flushReq** - description of flush (level of PTE, PTE, count) **flags -** new protection flags	**None** (void function).	Applies new protections to the BPN specified, if it could be translated to a PPN. Flushes TLBs as necessary. Effectively, updates the EPT tree to reflect new protections for a page.
GPhys_Validate	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**paddr** - physical address in page **errorCode** - page fault error code resulting in validation attempt **guestAccess** - whether the fault resulting in validation was incurred while running the guest **respectTraces** - whether to fire software callbacks for the page access	**Bool** - whether validation succeeded. If true, a mapping in the EPT tree now exists for the given paddr.	Main function to "validate" (create a new **PA (PPN)** => **MPN** mapping in the active EPT tree). Translates from PA to PPN to BPN to MPN, respecting and applying permissions at all levels. If all goes well, inserts the new translation into the EPT tree. Otherwise, fails and returns. Attempts to validate at the largest size/highest level applicable (to allow an invalidated set of small pages to become a singular large page thereafter).
GPhysInvalidateMappingRange	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**startPPN** - first PPN in range **endPPN** - last PPN in range	**None** (void function).	Invalidates all PPNs in the range at all relevant levels in the EPT tree. (Clears mappings for all PPNs specified)

Module Function	Security Function (s)	SFR(s)	Parameters	Return Value	Rationale
GPhys_InvalidatePageList	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**pageList** - list of pages to invalidate **numEntries** – list length (or 1) **isLargePages** - does the list represent 4KB pages or larger?	**None** (void function).	Invalidates a list of pages but does not flush the TLB. TLB flushing is the caller's responsibility.
GPhys_InvalidateBPN	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**bpn** - BPN to invalidate **toLargeFlushReq** - should this flush be added to an ongoing large flush request (for later TLB flush)?	**None** (void function).	Invalidates the specified BPN in the EPT tree but does not flush TLBs (allowing more efficient batching of TLB flushes). TLB flushing is the caller's responsibility.
BusMemInvalidateCache	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**pageList** - list of pages to invalidate **req** - zap request structure describing pages and release **isMPNlist** - zap one entry or a list? **numEntries** – list length (or 1) **isLargePages** - does the list represent 4KB pages or larger?	**None** (void function).	Invalidates list of pages (calls both GPhys_InvalidatePageList() and VNPT_InvalidatePageList()) and causes VM-wide TLB flushes by invoking BusMemZapPageListCC on all VCPUs.
GPhys_ConvertToLargeOne	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**ppn** - the PPN to be converted **bpn -** the BPN expected to be visible at this PPN	0 or 1: the number of pages converted to large.	Converts 512 aligned 4 kilobyte pages with identical permissions to a single 2 megabyte mapping, or fails and does no harm.

Module Function	Security Function (s)	SFR(s)	Parameters	Return Value	Rationale
GPhys_FlushAllTLBs	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Synchronously flushes all TLBs on all VCPUs in the VM. Slow, comprehensive.
GPhysOpenLargeFlushReq	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Opens a large flush request, which will contain a set of one or more PPNs to flush from the TLB. Used to efficiently batch flushing of many pages.
GPhys_AddToLargeFlushReq	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**flushReq** - request to add to set (either one PPN or "flush all")	**None** (void function).	Adds one PPN (or "flush all" command) to the current to-be-flushed set.
GPhys_CloseLargeFlushReq	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Closes a large flush request.
GPhys_ProcessLargeFlushReq	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Processes a large flush request, flushing either all PPNs in the set or the entire TLB, on the current CPU.
HV_FlushNestedMappings	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Invalidates the EPTP on the current CPU.

4.2.4.2 Internal Interfaces of the Module (VNPT, for nested guest memory virtualization)

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
VNPTSetNPTE	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**snpte** - shadow nested page table entry to update **val** - value to set **level** - page table level in EPT tree	**None** (void function).	Sets an EPTE in an NPT shadow (nested EPT tree). Also used to clear entries.

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
VNPT_Update	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**startBPN** - first (possibly only) BPN to update **action** - description of update action (invalidate, add or remove software callback hooks for page) **isLargePage** - act on one 4 kilobyte page or an aligned region of 512 4 kilobyte pages (one 2 megabyte page)?	**Bool** - whether a flush is needed after the update.	Updates an existing entry in a VNPT shadow, for either a 4 kilobyte or 2 megabyte page. Callers must flush the TLB in the relevant ASID if TRUE is returned.
VNPT_InvalidatePageList	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**pageList** - list of pages to invalidate **numEntries** – list length (or 1) **isLargePages** - does the list represent 4KB pages or larger?	**None** (void function).	Invalidates a list of pages in an NPT shadow but does not flush the TLB. TLB flushing is the caller's responsibility.

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
VNPT_HandleNPF	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**la** - linear address being accessed when the nested page fault was incurred **pal** - input physical address: the physical address causing the fault **flags** - description of the type of memory access (e.g. read, write, page table access)	**An x86fault object pointer,** representing either a specific failure, or successful validation in the VNPT shadow (X86Fault_None or similar).	Attempts to handle a nested page fault while in nested guest execution via VVT. When successful, updates the VNPT shadow for the access.
HVFlushAllASIDs	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Invalidates all VPIDs on the current CPU.
VVTProcessVPID	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**Bool** - TRUE if a non-zero VPID was set or if the VPID control was disabled.	Sets the VPID field in the virtual CPU (based upon nested VMCS state), for future use in VT execution.
VNPT_FlushPhysical	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Flushes all shadow nested page table mappings from current CPU.

4.2.5 Used interfaces of other modules

Module Function	Module	Description	Parameters	Return Value	File
HV_SetNestedPagingRoot	4.1 VMM Hardware Virtualization	Sets nested paging root (EPTP) in VMCS for use in VT execution.	**l4MPN** - MPN corresponding to EPTP to populate.	**None**. (void function).	vmcore/monitor/vmm/hv/vt/hv-common.h
VmMemPf_GetPFrameMPN	9.5 VM volatile memory virtualization	Request the MPN backing the given BPN from the vmkernel, for memory-backed BusMem regions. (Called via 4.8: VMM/VMK interface)	**bpn** - BusMem page to translate to a machine page **mpn** - pointer to return the translated MPN (or INVALID_MPN on failure)	**VMKCall return**: VMK_OK on success, a failure code otherwise.	vmkernel/mem/vmmempf.c
VMKCall_GetPFrameMPN	4.8 VMM/VMK interface	Switch to the vmkernel and call VmMemPf_GetPFrameMPN()	**bpn** - BusMem page to translate to a machine page **mpn** - pointer to return the translated MPN (or INVALID_MPN on failure)	**VMKCall return**: VMK_OK on success, a failure code otherwise.	vmcore/public/x86/vmkernelVmcoreFuncs.h

4.2.6 Mapping to the Source Code

Function	Description	File
GPhysSetPTE	Sets a PTE in a GPhysTree (EPT tree).	vmcore/monitor/vmm/gphys/common/gphys.c
GPhysClearPTE	Clears a PTE in a GPhysTree (EPT tree). Also clears VMM software mapping ("NPTMap") of the EPT subtree wired by the previous PTE.	vmcore/monitor/vmm/gphys/common/gphys.c
GPhysProtectPPN	Applies new protections to the BPN specified, if it could be translated to a PPN. Flushes TLBs as necessary. Effectively, updates the EPT tree to reflect new protections for a page.	vmcore/monitor/vmm/gphys/common/gphys.c

Function	Description	File
GPhys_Validate	Main function to "validate" (create a new **PA (PPN)** => **MPN** mapping in the active EPT tree). Translates from PA to PPN to BPN to MPN, respecting and applying permissions at all levels. If all goes well, inserts the new translation into the EPT tree. Otherwise, fails and returns. Attempts to validate at the largest size/highest level applicable (to allow an invalidated set of small pages to become a singular large page thereafter).	vmcore/monitor/vmm/gphys/common/gphys.c
GPhysInvalidateMappingRange	Invalidates all PPNs in the range at all relevant levels in the EPT tree. (Clears mappings for all PPNs specified)	vmcore/monitor/vmm/gphys/common/gphys.c
GPhys_InvalidatePageList	Invalidates a list of pages but does not flush the TLB. TLB flushing is the caller's responsibility.	vmcore/monitor/vmm/gphys/common/gphys.c
GPhys_InvalidateBPN	Invalidates the specified BPN in the EPT tree but does not flush TLBs (allowing more efficient batching of TLB flushes). TLB flushing is the caller's responsibility.	vmcore/monitor/vmm/gphys/common/gphys.c
BusMemInvalidateCache	Invalidates list of pages (calls both GPhys_InvalidatePageList() and VNPT_InvalidatePageList()) and causes VM-wide TLB flushes by invoking BusMemZapPageListCC on all VCPUs.	vmcore/monitor/vmm/main/busmem.c
GPhys_ConvertToLargeOne	Converts 512 aligned 4 kilobyte pages with identical permissions to a single 2 megabyte mapping, or fails and does no harm.	vmcore/monitor/vmm/gphys/common/gphys.c
GPhys_FlushAllTLBs	Synchronously flushes all TLBs on all VCPUs in the VM. Slow, comprehensive.	vmcore/monitor/vmm/gphys/common/gphys.c

Function	Description	File
GPhysOpenLargeFlushReq	Opens a large flush request, which will contain a set of one or more PPNs to flush from the TLB. Used to efficiently batch flushing of many pages.	vmcore/monitor/vmm/gphys/common/gphys.c
GPhys_AddToLargeFlushReq	Adds one PPN (or "flush all" command) to the current to-be-flushed set.	vmcore/monitor/vmm/gphys/common/gphys.c
GPhys_CloseLargeFlushReq	Closes a large flush request.	vmcore/monitor/vmm/gphys/common/gphys.c
GPhys_ProcessLargeFlushReq	Processes a large flush request, flushing either all PPNs in the set or the entire TLB, on the current CPU.	vmcore/monitor/vmm/gphys/common/gphys.c
HV_FlushNestedMappings	Invalidates the EPTP on the current CPU.	vmcore/monitor/vmm/hv/vt/hv-vt.c
VNPTSetNPTE	Sets an EPTE in an NPT shadow (nested EPT tree). Also used to clear entries.	vmcore/monitor/vmm/hv/common/vnpt-common.h
VNPT_Update	Updates an existing entry in a VNPT shadow, for either a 4 kilobyte or 2 megabyte page. Callers must flush the TLB in the relevant ASID if TRUE is returned.	vmcore/monitor/vmm/hv/common/vnpt-common.h
VNPT_InvalidatePageList	Invalidates a list of pages in an NPT shadow but does not flush the TLB. TLB flushing is the caller's responsibility.	vmcore/monitor/vmm/hv/common/vnpt-common.h
VNPT_HandleNPF	Attempts to handle a nested page fault while in nested guest execution via VVT. When successful, updates the VNPT shadow for the access.	vmcore/monitor/vmm/hv/common/vnpt-common.h
HVFlushAllASIDs	Invalidates all VPIDs on the current CPU.	vmcore/monitor/vmm/hv/vt/hv-vt.c
VVTProcessVPID	Sets the VPID field in the virtual CPU (based upon nested VMCS state), for future use in VT execution.	vmcore/monitor/common/hv/vt/vvt.c
VNPT_FlushPhysical	Flushes all shadow nested page table mappings from current CPU.	vmcore/monitor/vmm/hv/common/vnpt-common.h

4.2.7 Appendix A: Bibliography for Intel Documentation References (EPT)

Document	Author / Company	Date	Notes
Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3C: System Programming Guide, Part 3	Intel Corporation www.intel.com	06/30/2022	Specific citations are in the detail table below. **Note:** Intel renumbers resources over time. These chapters and the SDM volume number are correct as of June 30, 2022.
Chapter name	**Content**		
(28) VMX Support for Address Translation	Explanation of EPT and related technologies		

4.2.7 Appendix B: Navigating Guest Memory Module Code

The code and header files implementing the module are used across multiple products and CPU architectures. Only a subset of the code is of relevance to the TOE. This table endeavors to simplify reading code and header files by explaining what is included and excluded from the TOE. Terminology clarifying the above documentation is also provided.

Term or token	Meaning	Included in TOE?
vmx86_server	Set to 1 if building ESX	Yes
VMX86_SERVER (CPP token)	#defined if building ESX	Yes
SERVER_ONLY()	Macro contents defined if building ESX	Yes
HOSTED_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_vmm	Set to 1 if building VMM	Yes
vmx86_ulm	Set to 0 if building VMM	No
ULM_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_release	Set to 1 if building for releases to customers	Yes
vmx86_debug	Set to 1 if building for debug builds	No
vmx86_devel	Set to 1 if building for internal developers	No
vmx86_vt	Set to 1 if building for VT support	Yes

Term or token	Meaning	Included in TOE?
vmx86_svm	Not relevant to VT support (AMD-specific)	No
vmx86_ept	Set to 1 if building for VT support	Yes
vmx86_npt	Set to 0 if building for VT support (AMD-specific)	No
VCPU_InGuestOperation()	Returns TRUE if the VCPU is running or emulating the nested guest	Yes
GPhys_HWMMUTreeInVMK()	FALSE for the TOE. Ignore any code in which this must be TRUE.	No
GPHYS_TREE_HWMMU	The 'GPhysTree' in code relevant to EPT. Other trees are used for supporting emulation.	Yes

4.3 VMM Host Interrupts IDT, APIC, MAP (SFR-ENFORCING)

The VMM Host Interrupts IDT APIC Map module (hereafter, "interrupt optimization module") implements optimized interrupt handling and inter-thread communication within a virtual machine. The module implements direct control and use of CPU interrupt controller hardware in a manner cooperative with the vmKernel (which normally controls CPU interrupt controller hardware).

A VM is comprised of virtual CPUs, each of which is implemented by a thread (hereafter a "world"). The Virtual Machine Monitor ("VMM") is a kernel-mode program which implements execution of a VM. A VM contains one VMM world per virtual CPU, and often the VMM worlds within a VM run concurrently, on different host PCPUs.

When the vmKernel's CPU scheduler (see 8.1: CPU Dispatcher) runs a VMM world, that VMM world takes control of the Interrupt Descriptor Table Register (hereafter "IDTR") in the CPU, causing subsequent interrupt activity to run VMM interrupt handler code instead of vmKernel interrupt handler code. When a VMM world is run, a thread-local variable is updated to contain the APIC Identifier of the physical CPU running the world. With direct control of the IDTR and awareness of the APIC Identifiers of each VMM world, VMM worlds within a VM can directly send Inter-Processor Interrupts (hereafter "IPIs") to one another, and VMM handler code will execute in response. When a VMM world discontinues running, it relinquishes control of the IDTR to the vmKernel.

Because the vmKernel and the VMM cooperatively share use of the IDTR and the interrupt controllers of physical CPUs, the two must implement a careful protocol to avoid non-interference. Hence, the interrupt optimization module is SFR-enforcing because it must carefully avoid allowing VMM (which runs proximate to the guest domain) instances from interfering with one another or other host software.

IPIs, as with all interrupts on x86 CPUs, are targeted via 8-bit vector numbers. The vmKernel and VMM dedicate specific vectors to specific purposes. Two vectors are reserved for VMM IPI use: the posted interrupt vector, and the general monitor IPI vector. The exact vectors for these purposes must be carefully communicated by the vmKernel to VMM and respected by both pieces of software.

4.3.1 Security Functionality (SF)

See the table in Section 4.3.2.

4.3.2 Security Functional Requirement (SFR)

Security Function (SF)	Security Function Requirement (SFR)	Rationale
SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	The TSF shall maintain a security domain for the execution of each virtual machine that protects the virtual machine from interference and tampering by untrusted subjects or subjects from outside the scope of the VM.

4.3.3 Provided TSFI

This module has no TSFI as it is an internal module and has no exposure to outside the TOE.

4.3.4.1 Internal Interfaces of the Module

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
ApicMap_InterruptVcpuid	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**v** - the Vcpuid of the VCPU to interrupt **ipiVec** - the Inter-Processor Interrupt Vector to use	**None** (void function).	Interrupts a specified VCPU within this VM using the vector given.
Platform_GetMonitorIPIVector	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	An interrupt vector	Gets the IPI vector reserved by the platform (vmKernel) for general monitor use.
Platform_GetHVIPIVector	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	An interrupt vector	Gets the IPI vector reserved by the platform (vmKernel) for posted interrupt/HV use.
VMKCall_VMKGetIntInfo	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**inData** - return pointer for kernel interrupt information	A VMKCall return status (VMK_OK on success).	Acquires interrupt information from the vmKernel, including IPI vectors for the monitor to use.

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
IDT_NullGate	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**regs** - exception frame provided when hardware raises an interrupt (including IPIs).	**None** (void function).	Monitor handler quickly called when an IPI is sent to a CPU running a VMM world.
WorldSaveApicMap	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**w** - world that is transitioning from running to not running	**None** (void function).	Unsets apicMap variable for this VMM world, as it is no longer running and thus cannot receive IPIs.
WorldRestoreApicMap	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**w** - world that is transitioning from running to not running	**None** (void function).	Sets apicMap variable to the physical CPU's APIC Id for this VMM world, as it about to run and can thus receive IPIs.
WorldArchSharedAreaVcpuInit	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**world** - VMM world being initialized.	**VMK_ReturnStatus** - VMK_OK on successful initialization, or an error otherwise.	Initializes pointer to world's apicMap variable for later population when the VMM world is run or finishes running.

4.3.4 Used interfaces of other modules

None.

4.3.5 Mapping to the Source Code

Function	Description	File
ApicMap_InterruptVcpuid	Interrupts a specified VCPU within this VM using the vector given.	vmcore/monitor/vmm/main/apicmap.c
Platform_GetMonitorIPIVector	Gets the IPI vector reserved by the platform (vmKernel) for general monitor use.	vmcore/monitor/common/public/platform_vmk.h

Function	Description	File
Platform_GetHVIPIVector	Gets the IPI vector reserved by the platform (vmKernel) for posted interrupt/HV use.	vmcore/monitor/common/public/platform_vmk.h
VMKCall_VMKGetIntInfo	Acquires interrupt information from the vmKernel, including IPI vectors for the monitor to use.	vmcore/public/x86/vmKernelVmcoreFuncs.h
IDT_NullGate	Monitor handler quickly called when an IPI is sent to a CPU running a VMM world.	vmcore/monitor/vmm/cpu/idt.c
WorldSaveApicMap	Unsets apicMap variable for this VMM world, as it is no longer running and thus cannot receive IPIs.	vmKernel/main/x86/world_int_arch.h
WorldRestoreApicMap	Sets apicMap variable to the physical CPU's APIC Id for this VMM world, as it about to run and can thus receive IPIs.	vmKernel/main/x86/world_int_arch.h
WorldArchSharedAreaVcpuInit	Initializes pointer to world's apicMap variable for later population when the VMM world is run or finishes running.	vmKernel/main/x86/world.c

4.3.6 Appendix A: Navigating Interrupt Optimization Module Code

The code and header files implementing the module are used across multiple products and CPU architectures. Only a subset of the code is of relevance to the TOE. This table endeavors to simplify reading code and header files by explaining what is included and excluded from the TOE. Terminology clarifying the above documentation is also provided.

Term or token	Meaning	Included in TOE?
vmx86_server	Set to 1 if building ESX	Yes
VMX86_SERVER (CPP token)	#defined if building ESX	Yes
SERVER_ONLY()	Macro contents defined if building ESX	Yes
HOSTED_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_vmm	Set to 1 if building VMM	Yes
vmx86_ulm	Set to 0 if building VMM	No

Term or token	Meaning	Included in TOE?
ULM_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_release	Set to 1 if building for releases to customers	Yes
vmx86_debug	Set to 1 if building for debug builds	No
vmx86_devel	Set to 1 if building for internal developers	No
VMM_BOOTSTRAP	Set to 0 for general VMM run-time	No
Files and directories containing arm64	Not relevant to Intel/x86 product	No

4.4 VMM Hot Path (SFR-NON-INTERFERING)

The Virtual Machine Monitor (hereafter "VMM") in the TOE executes the guest domain (hereafter "VM") using Intel's VT via the VMM Hardware Virtualization module (see 4.2: VMM Hardware Virtualization, hereafter the "HV module"). The HV module tends to run in a loop, entering guest execution via an "HV resume" followed by the guest execution exiting and returning to VMM, in the HV module's "HV exit" code path. The VMM Hot Path module (hereafter "Hot Path module") attempts to optimize handling of HV exit reasons and qualifications and quick return to HV resume.

The Hot Path module quickly checks for various common and easily-handled exit causes. For example, if a guest operating writes to the Virtual CPU's interrupt controller (the Advanced Programmable Interrupt Controller, known as the APIC), the exit qualification describes this write, and the hot path module calls virtual APIC update code to quickly update APIC state in the virtual CPU, then returns such that the HV module can HV resume. If the Hot Path module fails to quickly handle an HV exit (whether because it was an uncommon exit, or because handling failed due to some uncommon circumstance), the Hot Path module calls what is known as the "slow path", relying upon software emulation (see 4.5: VMM Instruction Emulation, hereafter "Instruction Emulation module") to handle the exit and return.

The Hot Path module implements separate functionality for handling HV exits while in nested guest execution (see 4.2 for an explanation of nested virtualization). Special cases for nested guest execution and its optimization necessitate different handling, both for functional correctness and performance reasons. Some common code is used in handlers for both the non-nested and nested guest fast paths for HV exit handling.

The Hot Path module uses various portions of VMM in support of its fast handling of HV exits, as well as the instruction emulation module.

Because the Hot Path module does not act upon state relevant to SFR enforcement and manipulates software state in service of handling HV exits, it is SFR-NON-INTERFERING. It is essentially glue between modules within VMM, optimized for performance.

4.4.1 Mapping to the Source Code

Function	Description	File
HVTryFastExit	Attempts to quickly handle an HV exit. If the exit can be quickly handled, finishes with HV Resume. If the exit cannot be handled, returns, such that the caller (HVExit) can call the slow path.	vmcore/monitor/vmm/hv/vt/hv-common.h
HVTryFastNestedExit	Attempts to quickly handle an HV exit raised while running a nested guest. If the exit can be quickly handled, finishes with HV Resume. If the exit cannot be handled, returns, such that the caller (HVExit) can call the slow path.	vmcore/monitor/vmm/hv/vt/hv-common.h

4.5 VMM Instruction Emulation (SFR-NON-INTERFERING)

The Virtual Machine Monitor (hereafter "VMM") implements Virtual CPUs to run Virtual Machines. Virtual CPUs are implemented in accordance with Intel specifications for CPUs. Execution of a Virtual Machine is performed via the VMM Hardware Virtualization module (see 4.2, hereafter "HV module") utilizing Intel's VT. HV implements much of a correct and fast virtual CPU. Occasionally, VMM must emulate an instruction (as HV might otherwise infinitely loop, blocked by some attribute imposed upon VM execution by VMM). For example, if VMM requests a software callback before VM access to a particular page of memory, Intel's VT will exit to VMM (via an HV exit), and re-entry (via an HV resume) would simply result in another, identical exit. The Virtual CPU would not move forward. To unblock such instances, VMM contains an instruction emulation engine.

The VMM Instruction Emulation module (hereafter "Emulation module") implements the capability to emulate any valid instruction in any valid situation with any valid arguments, within a virtual CPU. Instruction emulation is slow, as it is a software implementation of all steps a CPU would perform to run an instruction (including complex instructions), and emulation also causes all correct results, effects and side-effects of the instruction's effective execution. Emulating a single instruction might take 1000 times as long in software emulation as in HV execution. The Emulation module implements decoding of an instruction including all instruction prefixes, execution of the instruction including emulation of any relevant guest register or memory accesses, potential faulting if execution dictates a fault be raised and possible indirect software side-effects.

To decode an instruction, the Emulation module contains an x86 instruction decoder aware of all possible instructions and their encodings and meanings. The Emulation module contains code called the interpreter which can "interpret" a single instruction, once decoded. The interpreter implements emulation of every valid instruction encoding. Memory accessed by instructions is handled by accesses to the VMM-mapped view of guest memory provided 4.2: VMM HV Memory Management. Software state of the virtual CPU, from general-purpose to special-purpose registers may be read or updated by the interpreter. The interpreter may call out to supporting functionality related to, for example, privileged instruction state (i.e., instructions which run in the more privileged guest kernel mode, as opposed to guest user

mode). As many instructions use different parts of the virtual CPU, various different functions provided by different code files may be called.

The Emulation module also contains a fast emulation engine known as HVSimulate. HVSimulate is a system to more quickly emulate one or more instructions (up to a maximum of twelve) from a small lexicon of simple and well-defined instructions. HVSimulate creates and manages a cache of instructions at which HV exits repeatedly occur, and at the third occurrence of such an exit, HVSimulate creates a "translation" of the instruction. This translation allows the Emulation module to amortize the cost of decoding an instruction, and to create an executable equivalent portion of emulation code to re-use when such an instruction is encountered again. HVSimulate then potentially builds chains of consecutive instructions (from its limited lexicon, with various simplifying constraints) to create larger translations. The HVSimulate engine within the Emulation module is the entrypoint to the Emulation module from the VMM Fast Path module. If HVSimulate lacks a translation (as the instruction may never have been seen before, or it is invalid for translation), it falls back to the interpreter, which is guaranteed to succeed in instruction emulation.

The Emulation module operates purely on software, virtual CPU state within the VMM program, to ensure forward progress of the virtual CPU as part of "slow path" operation. As such, it is SFR-NON-INTERFERING.

4.5.1 Mapping to the Source Code

Function	Description	File
HVSim_Try	Attempts to emulate one or more instructions efficiently instructions using HVSimulate. Run-time entrypoint to HVSimulate.	vmcore/monitor/common/hv/hvsimulate.c
Interp_Step	Emulates one guest instruction, delivering any resulting faults to the guest. Run-time entrypoint to the interpreter.	vmcore/monitor/common/cpu/x86/interp.c
Decoder_DecodeAtVCPU	Decodes the next guest instruction at the VCPU's current program counter. Run-time entrypoint to the interpreter for emulation of an instruction.	vmcore/monitor/common/cpu/x86/decoderMonitor.c

4.5.2 Appendix A: Published Technical Research Bibliography

Document	Author / Company	Date	Notes
Software Techniques for Avoiding Hardware Virtualization Exits	Ole Agesen, Jim Mattson, Radu Rugina, Jeffrey Sheldon, VMM team, VMware	2012 USENIX Annual Technical Conference, June	Describes the techniques used in HVSimulate for workloads of relevance in the year 2012. Workloads have changed but much of the paper is still accurate and applicable to HVSimulate code in the TOE.

4.6 VMM Guest Interrupts (SFR-NON-INTERFERING)

Virtual guest interrupts (hereafter "virtual interrupts") are a software construct implemented in virtual hardware. The Virtual Machine Monitor (hereafter "VMM") implements virtual interrupts in the VMM Guest Interrupt module (hereafter "Interrupt module"). A virtual machine (hereafter "VM") uses a virtual interrupt controller (hereafter "virtual APIC" or "VAPIC") to program interrupt behavior in a virtual CPU. The Interrupt module implements the virtual interrupt controller interface, its behavior, and the delivery of interrupts to the VM.

Virtual interrupts are asynchronous events which alter execution within a VM's software under various conditions. A virtual device, for example, may raise a virtual interrupt for delivery. One VCPU may request, via the virtual interrupt, an Inter-Processor Interrupt (hereafter "IPI") on another VCPU within the same VM. Regardless of the interrupt source, the Interrupt module implements VAPIC as well as interrupt delivery to the VM.

When entering VT execution (see 4.2: VMM Hardware Virtualization, hereafter "HV module"), the VMCS contains an interrupt state for the VCPU. If the interrupt state encodes that an interrupt is to be raised, VT will accordingly alter VM execution, such that the VCPU appropriately reacts to the interrupt and can handle it. The VM's software likely then interacts with the CPU and APIC to acknowledge, handle and end handling of the interrupt. The HV module also allows for an optimized form of external interrupt injection without the need to exit from VT execution, via "posted interrupts". The Interrupt module relies upon the HV module for this functionality.

All virtual interrupt controller state, all virtual devices capable of raising virtual interrupts and all CPU functionality related to virtual interrupts is VM-local, software-implemented and self-contained within a VMM instance. As such, the Interrupt module is SFR-NON-INTERFERING.

4.6.1 Mapping to the Source Code

Function	Description	File
APICSendInterrupt	Cause a virtual interrupt to be delivered to the a specified Virtual CPU.	vmcore/monitor/common/intr/x86/apic.c
APIC_PostIntr	Delivers a virtual interrupt to a remote Virtual CPU using posted interrupts (such that the remote VCPU may not exit VT at all to receive this interrupt).	vmcore/public/x86/apic_shared.h

4.7 VMM Timekeeping (SFR-NON-INTERFERING)

The perception of time is important in operating system and application software, whether run on a physical machine or in a virtual machine. While physical CPUs and platforms implement rigid, precise timekeeping, virtual CPUs and platforms run with additional

overhead, due to both virtualization overheads and time-shared scheduling. Operating systems and application software nonetheless have requirements and expectations, regardless of whether run physically or virtually.

The VMM Timekeeping module (hereafter "Timekeeping module") accommodates the time-perception needs of software within a VM. VM software largely perceives time via reads of the CPU cycle clock (using the RDTSC instruction or variants thereof) and by the VM's interaction with virtual timer hardware.

A VM's cycle clock runs at a constant rate based upon the CPU clock rate evident to the VM at the time it powered on, based upon the underlying physical CPU's clock rate. Thereafter, even if the VM migrates to another host with a different physical CPU clock rate, the VM will maintain the perception of the original clock rating. This requires scaling of reads of the TSC, which the Timekeeping module must provide.

The virtual CPU cycle clock must also be monotonic across all VCPUs within a VM: successive reads must always yield increasing values. The Timekeeping module must provide this.

Ideally, virtualization overheads related to timekeeping should be minimized, for performance. Thus the Timekeeping module endeavors to avoid overheads when the VM reads the VCPU's cycle clock, by use of VT controls to automatically scale or offset RDTSC responses without incurring an HV exit. (For VT control and HV exit discussion, see 4.1: VMM Hardware Virtualization).

Operating system software programs hardware timers to fire either periodically or at fixed times in the future. Virtual hardware implementing such timers must overcome additional challenges as compared to a physical host, because virtualization incurs additional overheads and because it is possible that a timer for a VM would fire while the VM is not scheduled by the host operating system. It is the Timekeeping module's responsibility to mitigate virtualization overheads and to endeavor to cause virtual timer interrupts to fire with approximately the accuracy expected on physical hosts.

Virtual hardware which indirectly relies upon timers must also behave coherently with overtly VM-visible timers.

To satisfy a need for coherent timers and timer-based operations, both visible to the VM directly via virtual hardware timer devices and indirectly via other means, the Timekeeping module implements a system called TimeTracker. This system provides a virtual clock source and a notion of "apparent time" to the VM. Various clients draw upon this same clock source.

Because virtual time is entirely confined to an individual VM and not exposed to other VMs or host software, the Timekeeping module is SFR-NON-INTERFERING.

4.7.1 Mapping to the Source Code

Function	Description	File
TimeTracker_ApparentTime	Calculates and returns a current "apparent time" in units of CPU cycles. Used by various devices and other portions of TimeTracker.	vmcore/monitor/common/main/x86/timeTracker.c

4.8 [vmKernel] VMM-VMK (SFR- ENFORCING)

The VMM-VMK Entry module manages run-time edges between execution of one thread of the Virtual Machine Monitor (hereafter "VMM") and the vmKernel. Two types of switches are managed: the start and end of run of a VMM thread (hereafter "VMM world" in VMware nomenclature), and the switch within a VMM world between the vmKernel context and the VMM context.

The vmKernel's CPU dispatcher (see 8.1: CPU dispatcher) causes worlds to run. When a world is run or finishes running, a context switch is affected (hereafter, "world switch") between the previously-running world and the next world to run. World switching saves CPU state of the previous world into memory and then loads state from memory to the CPU for the next world. For performance and correctness, VMM worlds contain specialized state which must be switched in addition to regular world-switch. The VMM-VMK Entry module implements this extra world switching for VMM worlds. Not only is some state saved and loaded, but some state is also flushed. Some of this switched state is relevant to SFR-enforcement.

Within a VMM world, two contexts exist: the vmKernel context and the VMM context. The VMM context takes nearly full control of the CPU and cooperatively shares the CPU with the vmKernel context. The VMM context enters the vmKernel context by affecting a specialized form of call (hereafter "VMKCall"). The VMM is re-entered from the vmKernel when a VMKCall returns. The VMM-VMK Entry module implements VMKCalls and returns from VMKCalls.

The VMM-VMK Entry module optimizes performance by minimizing the frequency and timing of some necessary operations. World switches are far less common than VMKCalls and VMKCalls are less common than HV Exits (see 4.1: VMM Hardware Virtualization). Many operations must be performed before leaving the VMM context or before world-switching to another world. Such operations can potentially be deferred to less common code paths (e.g. VMKCall, world switch).

Intel CPUs implement some functionality by use of Model-Specific Registers (hereafter "MSRs"). Some MSRs are benign, regardless of value, in some contexts. For example, in kernel-level software, the MSRs related to user-level system call handling are not relevant and can contain any valid value. Only when the CPU may run user-level software is it important to ensure that such MSRs contain appropriate values. To optimize VMM performance (recall that VMM is a kernel-level program), the VM is sometimes allowed read/write access to such MSRs, and the MSR values stay loaded into the CPU until world-

switch, at which time appropriate values are loaded. This functionality is managed by the MonMSR portion of the module.

Intel CPUs also contain special state related to virtualization, which is only relevant while running VMM worlds. As such, non-VMM worlds do not need to flush or reload this state - it is benign during execution of non-VMM worlds. For performance reasons, some virtualization-related CPU state is not reload or flushed until a new VMM world runs on a CPU. In VMware nomenclature, a CPU is considered "tainted" if the last VMM world to run on the CPU is not the current VMM world. The VMM-VMK Entry module reloads and/or flushes relevant CPU resources before their use by a new VMM world, if the CPU is tainted. (Note that if a CPU runs a VMM world, followed by a non-VMM world, followed by the initial VMM world again, no tainting occurs and there is no need for reloading/flushing).

Because the benign CPU state is loaded while running VMM or other host software, the VMM-VMK Entry module is SFR-Enforcing. The module is responsible for guaranteeing that such CPU state is benign, and that reloads and/or flushes occur to shield other software in the TOE from interference.

The VMM-VMK Entry module indirectly facilitates calls from VMM to the VMX. The VMM contains a mechanism called a "UserCall" which causes the VMKCall "VMKCall_SwitchToVCPU" which in turn requests that the vmkernel world-switch to a VMX world within the program.

4.8.1 Security Functionality (SF)

See the table in Section 4.8.2.

4.8.2 Security Functional Requirement (SFR)

Security Function (SF)	Security Function Requirement (SFR)	Rationale
SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	The TSF shall maintain a security domain for the execution of each virtual machine that protects the virtual machine from interference and tampering by untrusted subjects or subjects from outside the scope of the VM.

4.8.3 Provided TSFI

This module has no TSFI as it is an internal module and has no exposure to outside the TOE.

4.8.4.1 Internal Interfaces of the Module (World-Switch: Model-Specific Registers)

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
MonMSR_Init	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Initializes switchedMSRs values to be used during world switch. Loads initial monitor MSR values.
MonMSRInitSwitchedMSRs	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Initializes switchedMSRs values to be used during world switch.
MonMSR_SaveHostLoadMonitorState	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Loads initial monitor MSR values.
MonMSR_LoadMonitorState	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**None**	**None** (void function).	Loads initial monitor MSR values.
MonMSR_LoadMonitorMSR	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**msr** - the model-specific register to write into **val** - the value to write	**None** (void function).	Loads one monitor MSR value (if not masked).
MonMSRLoadMSR	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**msr** - the model-specific register to load **loadMask** - mask determining whether a CPU load should be affected for MSRs. **val** - the value to write	**None** (void function).	Loads one monitor MSR value (if not masked).

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
MonMSR_SetMSRUnused	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**msr** - the model-specific register to mark as unused	**None** (void function).	Marks an MSR as not used by the monitor (such that the monitor considers any value benign and world switch will not reload monitor values for this MSR).
MonMSR_SetMSR	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**msr** - the model-specific register to set **newVal** - the monitor value to set in the MSR **flags** - flags related to switching of the MSR	**None** (void function).	Marks an MSR as used by the monitor, setting its value and flags related to switching. If the value or flags have changed or if the flags specify that the MSR is not shadowed, its value is also loaded.
MonMSR_UpdateMSR	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**msr** - the model-specific register to set **newVal** - the (new) monitor value to set in the MSR	**None** (void function).	The value of the specified monitor-used MSR is updated (but flags are left unchanged). If the existing flags specify that the MSR is in use, the new value is loaded.
WorldSaveFastMSRs	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**save** - world being saved (transitioning from running to not running)	**None** (void function).	Saves monitor MSRs according to flags.

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
WorldRestoreFastMSRs	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**restore** - world being restored (transitioning from not running to running)	**None** (void function).	Loads monitor MSRs according to flags.
WorldArchSharedAreaVcpuInit	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**world** - VMM world being initialized.	**VMK_ReturnStatus** - VMK_OK on successful initialization, or an error otherwise.	Initializes pointer to world's switchedMSR variable for later use during world switch

4.8.4.2 Internal Interfaces of the Module (World-Switch: VT State)

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
WorldSaveControlRegisters	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**save** - world being saved (transitioning from running to not running)	**None** (void function).	Saves CPU control registers and "saves" VT state (flushes any active VMCS).
WorldSaveVTState	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**world** - world being saved.	**None** (void function).	"Saves" VT state (flushes any active VMCS).
WorldRestoreControlRegisters	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**restore** - world being restored (transitioning from not running to running)	**None** (void function).	Restores CPU control registers and restores VT state.
WorldRestoreVTState	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**world** - world being restored.	**None** (void function).	Restores VT state (loads any active VMCS).
World_ArchExit	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**w** - world that is transitioning from running to not running	**None** (void function).	"Saves" VT state (flushes any active VMCS) one final time before the world exits permanently.

4.8.4.3 Internal Interfaces of the Module (VMKCall: State Flushing)

Module Function	Security Function(s)	SFR(s)	Parameters	Return Value	Rationale
VMKCallWork	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**fnNum** - function number specifying which VMKCall to execute in the vmKernel context **args** - a pointer to arguments to the VMKCall	**VMK_ReturnStatus** - a return status for a VMKCall. Generally VMK_OK on success and a different error value on failure.	Switches from the VMM context to the vmKernel context to request the specified call.
SwitchRootAndBackTovmKernel	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**fnNum** - function number specifying which VMKCall to execute in the vmKernel context **args** - a pointer to arguments to the VMKCall **eflags** - CPU flags register value in VMM	**VMK_ReturnStatus** - a return status for a VMKCall. Generally VMK_OK on success and a different error value on failure.	Inner function that switches from the VMM context to the vmKernel context to request the specified call. May never return, and may also return running on a new CPU (with 'pcpuTainted' set).
HV_EnterMonitor	SF6.Protection of the TSF (FPT)	FPT_VIV_EXT.1.1	**pcpuTainted** - is the current CPU tainted (i.e. this VMM world was not the last world to run on the CPU, and flushing is needed)?	**None** (void function).	Called upon re-entry to the monitor to restore HV state of relevance.If pcpuTainted is set, affects flushes of the TLB for VNPT, EPTP and all ASIDs (see 4.2: VMM HV Memory Management).

4.8.5 Used interfaces of other modules

Module Function	Module	Description	Parameters	Return Value	File
HV_FlushNestedMappings	4.2: VMM HV Memory Management	Invalidates the EPTP on the current CPU.	**None**	**None** (void function).	vmcore/monitor/vmm/hv/vt/hv-vt.c
VNPT_FlushPhysical	4.2: VMM HV Memory Management	Flushes all shadow nested page table mappings from current CPU.	**None**	**None** (void function).	vmcore/monitor/vmm/hv/common/vnpt-common.h
HVFlushAllASIDs	4.2: VMM HV Memory Management	Invalidates all VPIDs on the current CPU.	**None**	**None** (void function).	vmcore/monitor/vmm/hv/vt/hv-vt.c

None.

4.8.6 Mapping to the Source Code

Function	Description	File
MonMSR_Init	Initializes switchedMSRs values to be used during world switch. Loads initial monitor MSR values.	vmcore/monitor/vmm/main/monMSR.c
MonMSRInitSwitchedMSRs	Initializes switchedMSRs values to be used during world switch.	vmcore/monitor/vmm/main/monMSR.c
MonMSR_SaveHostLoadMonitorState	Loads initial monitor MSR values.	vmcore/monitor/vmm/main/monMSR.c
MonMSR_LoadMonitorState	Loads initial monitor MSR values.	vmcore/monitor/vmm/main/monMSR.c
MonMSR_LoadMonitorMSR	Loads one monitor MSR value (if not masked).	vmcore/monitor/vmm/main/monMSR.c

Function	Description	File
MonMSRLoadMSR	Loads one monitor MSR value (if not masked).	vmcore/monitor/vmm/main/monMSR.c
MonMSR_SetMSRUnused	Marks an MSR as not used by the monitor (such that the monitor considers any value benign and world switch will not reload monitor values for this MSR).	vmcore/public/monMSR.h
MonMSR_SetMSR	Marks an MSR as used by the monitor, setting its value and flags related to switching. If the value or flags have changed or if the flags specify that the MSR is not shadowed, its value is also loaded.	vmcore/public/monMSR.h
MonMSR_UpdateMSR	The value of the specified monitor-used MSR is updated (but flags are left unchanged). If the existing flags specify that the MSR is in use, the new value is loaded.	vmcore/public/monMSR.h
WorldSaveFastMSRs	Saves monitor MSRs according to flags.	vmkernel/main/x86/world.c
WorldRestoreFastMSRs	Loads monitor MSRs according to flags.	vmkernel/main/x86/world.c

Function	Description	File
WorldArchSharedAreaVcpuInit	Initializes pointer to world's switchedMSR variable for later use during world switch	vmkernel/main/x86/world.c
WorldSaveControlRegisters	Saves CPU control registers and "saves" VT state (flushes any active VMCS).	vmkernel/main/x86/world.c
WorldSaveVTState	"Saves" VT state (flushes any active VMCS).	vmkernel/main/x86/world.c
WorldRestoreControlRegisters	Restores CPU control registers and restores VT state.	vmkernel/main/x86/world.c
WorldRestoreVTState	Restores VT state (loads any active VMCS).	vmkernel/main/x86/world.c
World_ArchExit	"Saves" VT state (flushes any active VMCS) one final time before the world exits permanently.	vmkernel/main/x86/world.c
VMKCallWork	Switches from the VMM context to the vmKernel context to request the specified call.	vmcore/monitor/vmm/platform/vmkernel/vmk_if.c
SwitchRootAndBackTovmKernel	Inner function that switches from the VMM context to the vmKernel context to request the specified call. May never return, and may also return running on a new CPU (with 'pcpuTainted' set).	vmcore/monitor/vmm/platform/vmkernel/vmk_if.c

Function	Description	File
HV_EnterMonitor	Called upon re-entry to the monitor to restore HV state of relevance. If pcpuTainted is set, affects flushes of the TLB for VNPT, EPTP and all ASIDs (see 4.2: VMM HV Memory Management).	vmcore/monitor/vmm/hv/vt/hv-vt.c

4.8.7 Appendix A: Navigating VMM-VMK Entry Module Code

The code and header files implementing the module are used across multiple products and CPU architectures. Only a subset of the code is of relevance to the TOE. This table endeavors to simplify reading code and header files by explaining what is included and excluded from the TOE. Terminology clarifying the above documentation is also provided.

Term or token	Meaning	Included in TOE?
vmx86_server	Set to 1 if building ESX	Yes
VMX86_SERVER (CPP token)	#defined if building ESX	Yes
SERVER_ONLY()	Macro contents defined if building ESX	Yes
HOSTED_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_vmm	Set to 1 if building VMM	Yes
vmx86_ulm	Set to 0 if building VMM	No
ULM_ONLY()	Not relevant to ESX, enclosed contents omitted	No
vmx86_release	Set to 1 if building for releases to customers	Yes
vmx86_debug	Set to 1 if building for debug builds	No
vmx86_devel	Set to 1 if building for internal developers	No
VMM_BOOTSTRAP	Set to 0 for general VMM run-time	No
Files and directories containing arm64	Not relevant to Intel/x86 product	No
World_IsVMMWorld()	True if the world is a VMM world	Yes
World_IsKLMWorld()	Always FALSE for TOE.	No

Term or token	Meaning	Included in TOE?
HVMSR*	An MSR-switching subsystem not used for the TOE.	No
MonMSRLoadHostState	Not used in the TOE	No
MonMSRSaveHostState	Not used in the TOE	No
MonMSRLoadHostMSR	Not used in the TOE	No
switchedMSRs	Variable referenced via shared memory: world-switched MSRs.	Yes

4.9 VMM SGX (SFR-NON-INTERFERING)

SGX is a CPU feature provided by Intel enabling workloads to run within an execution context known as a Secure Enclave, provided by the CPU. Software outside a Secure Enclave (including operating system software, hypervisor software and even guest kernel and user mode software) cannot examine with or tamper with software or date inside the Enclave. SGX Enclaves run within encrypted, protected memory known as an Enclave Page Cache (EPC). SGX enclaves are created using EPC memory acquired via coordination with system firmware, and then bootstrapped and entered using SGX-specific instructions in the CPU.

The VMM provides Virtual SGX (hereafter "VSGX") to Virtual Machines. VSGX behaves as SGX, and is implemented using the same EPC memory and the same instruction-level interfaces provided by physical CPUs for SGX use. The SGX module implements VSGX.

To acquire EPC memory, the SGX module relies upon the VMM HV Memory Management module (see 4.2). EPC memory is separate from but largely handled in the same manner as non-volatile memory by the VMM HV Memory Management module. Virtual firmware exposes EPC memory to VM software much as physical firmware exposes EPC memory to operating system software on physical systems.

Most of the time, VSGX instructions and enclave code run directly on hardware without exiting to VMM, but occasionally it is necessary to emulate an SGX instruction. To emulate SGX instructions, the SGX module implements SGX-specific emulation functions which are in turn called by the VMM Instruction Emulation module (see 4.5) as necessary.

SGX-related CPU state is world-switched by 4.8 (VMM-VMK Entry module).

Because the SGX module uses EPC memory provided by the VMM HV Memory Management module and because this memory and SGX instructions are used for VM-private execution (with no interaction with other VMs nor other host software), the SGX module is SFR-NON-INTERFERING.

4.9.1 Mapping to the Source Code (Interpreter support)

Function	Description	File
SGX_Init	Initializes VMM's SGX functionality. Allows VSGX use by the VCPU thereafter.	vmcore/monitor/vmm/main/sgx_monitor.c
Interp_ENCLV	Interprets an ENCLV exit, emulating the requested leaf function.	vmcore/monitor/vmm/cpu/interpSGX.c
Interp_ENCLS	Interprets an ENCLS exit, emulating the requested leaf function.	vmcore/monitor/vmm/cpu/interpSGX.c

4.9.2 Appendix A: Bibliography for the Intel SGX References

Document	Author / Company	Date	Notes
Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3D: System Programming Guide, Part 4	Intel Corporation www.intel.com	07/27/2022	Describes SGX in great detail.

Confluence Concordance (VMware internal use)

Section	Page Version	Page Link
4	8-4-2022	VMM Subsystem - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.1	8-1-2022	4.1 VMM Hardware Virtualization - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.2	8-4-2022	4.2 VMM HV Memory Management - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.3	8-1-2022	4.3 VMM Host interrupts IDT APIC Map - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.4	8-1-2022	4.4 VMM Hot Path - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.5	8-2-2022	4.5 VMM Instruction Emulation - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.6	8-3-2022	4.6 VMM Guest Interrupts - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.7	8-3-2022	4.7 VMM Timekeeping - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.8	8-4-2022	4.8 [vmKernel] VMM-VMK Entry - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence
4.9	8-3-2022	4.9 VMM SGX - vSphere Certification - VMware Core Confluence - vSphere Certification - VMware Core Confluence