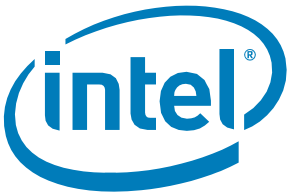

Guest-Host-Communication Interface (GHCI) for Intel® Trust Domain Extensions (Intel® TDX) 1.5

348552-001US

SEPTEMBER 2021

Disclaimers

Intel Corporation ("Intel") provides these materials as-is, with no express or implied warranties.

All products, dates, and figures specified are preliminary, based on current expectations, and are subject to change without notice. Intel does not guarantee the availability of these interfaces in any future product. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

The products described might contain design defects or errors known as errata, which might cause the product to deviate from published specifications. Current, characterized errata are available on request.

Intel technologies might require enabled hardware, software, or service activation. Some results have been estimated or simulated. Your costs and results might vary.

No product or component can be absolutely secure.

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted that includes the subject matter disclosed herein.

No license (express, implied, by estoppel, or otherwise) to any intellectual-property rights is granted by this document.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Copies of documents that have an order number and are referenced in this document or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting http://www.intel.com/design/literature.htm.

Table of Contents

1 About this Document

1.1 Scope of this Document

Trust Domains (TDs) are used to enable confidential hosting of VM workloads that are hardware-isolated from the hosting VMM and service OS environments. The Intel® Trust Domain Extensions (Intel® TDX) architecture enables isolation of the TD-CPU context and memory from the hosting environment. This document specifies the guest (TD) to host (VMM) communication interface that will be utilized for the paravirtualization interface between the TD and the VMM. This approach helps the Intel TDX-architecture prevent the VMM from accessing any TD runtime state. Hence, the TD must volunteer information to access IO services, enumerate model-specific, CPU capabilities, measurement services, and provide feedback to the VMM on guest-OS-triggered actions, such as virtual-IPIs, shutdown, etc. For each operation in this interface, the recommended actions are described for the host VMM (informative). The TD and the VMM are designed to use the subfunctions, which are normative and described in this document.

This document is a work in progress and is subject to change based on customer feedback and internal analysis. This document does not imply any product commitment from Intel to anything in terms of features and/or behaviors.

1.2 Document Organization

In Section 2, the document describes a general structure/ABI of the instruction TDCALL with the TDG.VP.VMCALL leaf used for passing information to the VMM and receiving information from the VMM. Section 3 describes the sub-leaves of TDCALL [TDG.VP.VMCALL] that define the ABI between the TD and the VMM for specific operations. Section 4 describes example flows for the main scenarios.

1.3 Glossary

Table 1-1: Intel TDX Glossary

Acronym	Full Name	Description
TME	**Total Memory Encryption**	An SoC memory encryption/decryption engine used to encrypt memory contents exposed externally from the SoC using an ephemeral, platform key. Memory is decrypted using the TME when memory contents are brought into the CPU caches.
MKTME	**Multi-Key TME**	This SoC capability adds support to the TME to allow software to use separate (one or more) keys for encryption of volatile- or persistent-memory encryption. When used with Intel TDX, it can provide confidentiality via separate keys for memory-used TDs. MKTME may be used with and without Intel TDX extensions.[1]

[1] In this document, the term "MKTME" means both the feature and the encryption engine itself.

Acronym	Full Name	Description
TD	**Trust Domain**	Software operating in a CPU mode designed to exclude the host/VMM software and untrusted, platform devices from the operational TCB for confidentiality. The operational TCB for a TD includes the CPU, the TD OS, and TD applications. A TD's resources are managed by an Intel TDX-aware-host VMM, but its state protection is managed by the CPU and is not accessible to the host software.
Intel TDX	**Intel TDX Architecture**	Intel-CPU-instruction-set-architecture extensions to enable host VMM to host Trust Domains
Intel TDX module	**Intel Trust Domain Extensions module**	Intel TDX module is a CPU-measured, software module that uses the instruction-set architecture for Intel TDX to help enforce security properties for hosting TDs on an Intel TDX platform. Intel TDX module exposes the Guest-Host-Communication Interface that TDs use to communicate with the Intel TDX module and the host VMM.
HKID	**Host Key ID**	When MKTME is activated, HKID is a key identifier for an encryption key used by one or more memory controller on the platform. When Intel TDX is active, the HKID space can be partitioned into a CPU-enforced space (for TDs) and a VMM-enforced space (for legacy VMs).
-	**TD-Private Memory (Access)**	TD-Private Memory can be encrypted by the CPU using the TD-ephemeral key (or in the future with additional, TD private keys).
-	**TD-Shared Memory (Access)**	TD-Shared Memory is designed to be accessible by the TD and the host software (and/or other TDs). TD-Shared Memory uses MKTME keys managed by the VMM for encryption.
TDVPS	**TD Virtual Processor Structure**	TD per-VCPU state maintained in protected memory by the Intel TDX.
TDCS	**Trust Domain Control Structure**	Multi-page-control structure for a TD. By design, TDCS is encrypted with the TD's ephemeral, private key, its contents are not architectural, and its location in memory is known to the VMM.

1.4 References

Table 1-2: Technical Documents Referenced

#	Reference Document	Version & Date
1	Intel® 64 and IA-32 Architecture Software Developer Manual	May 2020
2	Intel® Trust Domain Extensions CPU architecture specification	May 2021
3	Intel® Trust Domain Extensions module 1.5 base architecture specification	September 2021
4	Intel® Multi-key Total Memory Encryption (MK-TME) specification	April 2021
5	ACPI specification, version 6.4	January 2021
6	UEFI specification, version 2.9	March 2021
7	Intel® Trust Domain Extensions module 1.5 ABI reference specification	September 2021
8	Intel® Trust Domain Extensions module 1.5 architecture specification: TD Migration	September 2021

When specifying requirements or definitions, the level of commitment is specified following the convention of RFC 2119: Key words for use in RFCs to indicate Requirement Levels, as described in the following table:

Table 1-3: Requirement and Definition Commitment Levels

Keyword	Description
Must	“Must”, "Required", or "Shall" means that the definition is an absolute requirement of the specification.
Must Not	“Must Not” or "Shall Not" means that the definition is an absolute prohibition of the specification.
Should	“Should” or "Recommended" means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
Should Not	“Should Not” or the phrase "Not Recommended" means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood, and the case must be carefully weighed before implementing any behavior described with this label.
May	“May” or "Optional" means that an item is discretionary. An implementation may choose to include the item, while another may omit the same item because of various reasons.

2 TD-VMM Communication

2.1 Recap of Intel® Trust Domain Extensions (Intel® TDX)

Intel® Trust Domain Extensions (Intel® TDX) is an Intel technology that extends Virtual Machines Extensions (VMX) and Multi-Key Total Memory Encryption (MKTME) with a new kind of virtual machine guest called **Trust Domain (TD)**. A TD is designed to run in a CPU mode that protects the confidentiality of TD memory contents and the TD's CPU state from other software, including the hosting Virtual-Machine Monitor (VMM), unless explicitly shared by the TD itself.

The **Intel TDX module** uses the instruction-set architecture for Intel TDX and the MKTME engine in the SOC to help serve as an intermediary between the host VMM and the guest TDs. Details of the operation of this module are described in the Intel TDX-module specification [3]. The Intel TDX module exposes the **Guest-Host-Communication Interface (GHCI) for Intel TDX** (this specification) that TDs must use to communicate with the Intel TDX module and the host VMM.

As shown in the diagram below, an Intel TDX-aware, **host VMM** can launch and manage both guest TDs and legacy-guest VMs. The host VMM can maintain legacy functionality from the legacy VMs' perspective; the aim is for the host VMM to be restrict only with regard to the TDs it manages.

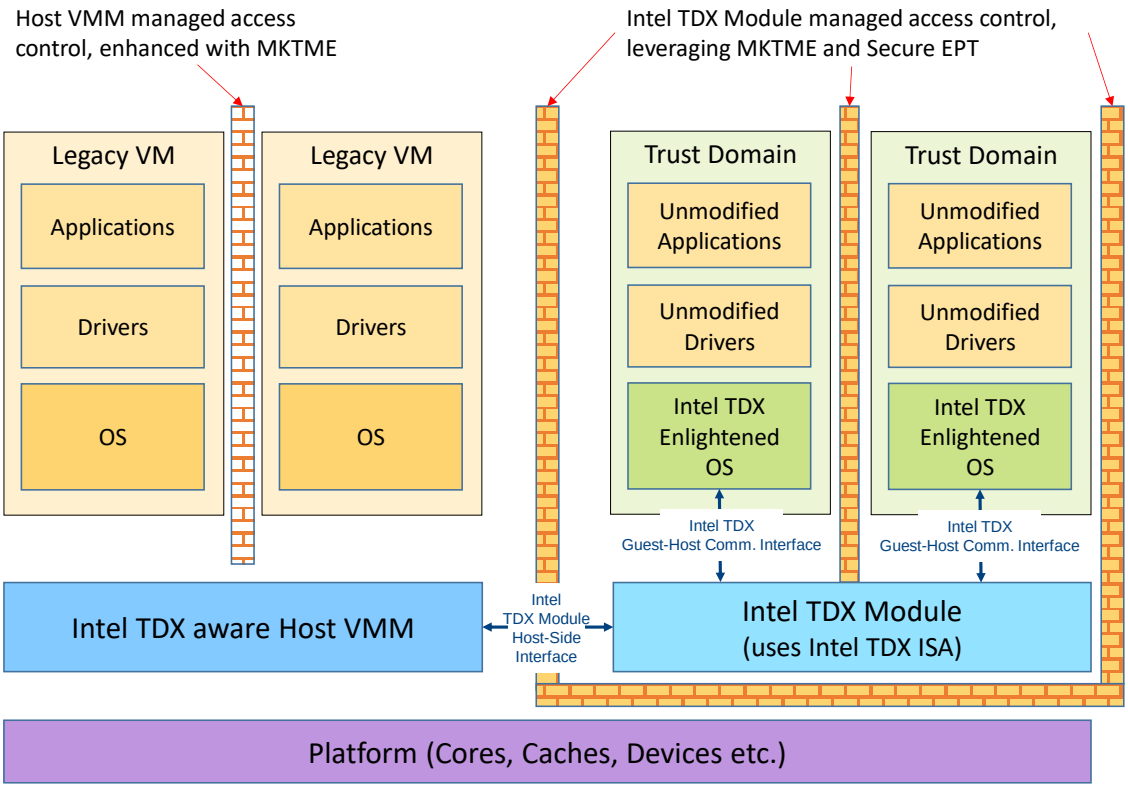

Figure 2-1: Components of Intel Trust Domain Extensions

2.2 TD-VMM-Communication Overview

TD-VMM communication can occur via asynchronous, VM exits or via synchronous (instruction), VM exits. In response to the synchronous (instruction), VM exits, Intel TDX [3] is designed to generate a Virtualization Exception (#VE) [1] for instructions the TD would be disallowed to invoke. The TD-guest software may respond by using the Intel TDX-provided information directly and/or after further decoding of the instruction that caused the #VE. The TD response must be via a TDCALL instruction [2] requesting the host VMM to provide (untrusted) services. The goal is for the VMM to receive the service request via a SEAMRET invoked by the Intel TDX module, complete the service requested, and respond to the TD via the SEAMCALL[TDH.VP.ENTER] to re-enter the TD. This document describes the mechanisms and ABI for this interaction in various scenarios expected.

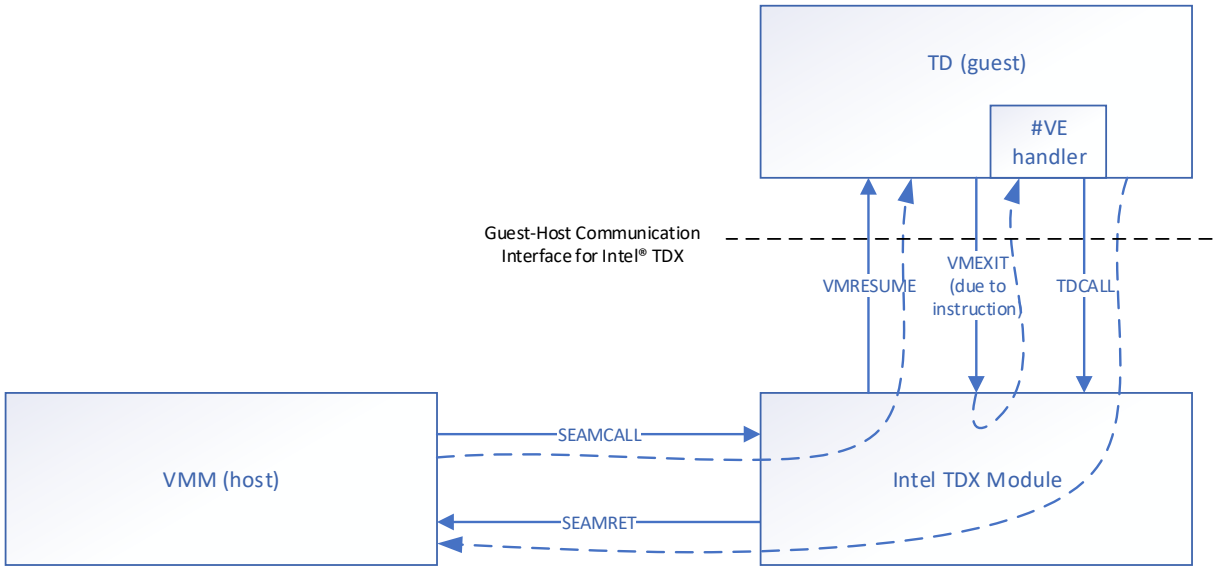

Figure 2-2: TD Guest-Host communication

Section 2 of this document describes the Virtualization Exception (#VE) for Intel TDX, and subsequent chapters describe the normative, TDCALL leaves intended to get the VE information as well as request services from the host VMM. There are other cases that may cause asynchronous, VM exits to the host VMM (via SEAMRET); for those scenarios, please refer to the Intel TDX module specification [3].

Section 3 of this document describes the reference/informative TDCALL[TDG.VP.VMCALL] interface sub-leaves intended to request services from the host VMM.

Section 4 describes the scenarios where TD-VMM communication interfaces described in this specification can be applied.

2.3 Virtualization Exception (#VE)

Intel TDX can cause #VE to be reported to the guest-TD software in cases of disallowed, instruction execution, *i.e.*, IO accesses, etc.

The goal is for #VE delivery by Intel TDX module to follow the architectural #VE handling for nested #VE, as described in Intel-SDM Chapter 25.5.6.3 (Delivery of Virtualization Exceptions). The TD OS should avoid instructions that may cause #VE in the #VE handler.

For detailed information about virtualization exception in TDX, please refer to Intel TDX Module Architecture specification.

2.4 TDCALL and SEAMCALL instruction

TDCALL is the instruction used by the guest TD software (in TDX non-root mode) to invoke guest-side TDX functions. For detailed information about TDCALL instruction, please refer to Intel TDX Module Architecture specification.

SEAMCALL is the instruction used by the host VMM to invoke host-side TDX functions. For detailed information about SEAMCALL instruction, please refer to Intel TDX Module Architecture specification.

2.4.1 TDCALL [TDG.VP.VMCALL] leaf

TDG.VP.VMCALL is a leaf function 0 for TDCALL. It helps invoke services from the host VMM. The input operands for this leaf are programmed as defined below:

Table 2-1: TDG.VP.VMCALL-Input Operands

Operand	Description
RAX	TDCALL instruction leaf number per Intel TDX Module Specification (0 - TDG.VP.VMCALL)
RCX	A bitmap that controls which part of the guest TD GPR and XMM state is passed as-is to the VMM and back. Please refer to Intel TDX Module Specification TDG.VP.VMCALL.
R10	Set to 0 indicates that TDG.VP.VMCALL leaf used in R11 is defined in this specification. All other values 0x1 to 0xFFFFFFFFFFFFFFFF indicate TDG.VP.VMCALL is vendor-specific (both R10 and R11)
R11	TDG.VP.VMCALL sub-function if R10 is 0 (see enumeration below)
RBX, RBP, RDI, RSI, R8-R10, R12–R15	See each TDG.VP.VMCALL sub-function for which registers must be used to pass values to the VMM (by setting RCX bits specified above)

Table 2-2: TDG.VP.VMCALL-Output Operands

Operand	Description
RAX	TDCALL instruction return code. Always returns Intel TDX_SUCCESS (0).
RCX	Unmodified
R10	**TDG.VP.VMCALL sub-function return value See table 2-6.** 0 – if no error Non 0 – if error happens. The error code is command specific.
R11	**See each TDG.VP.VMCALL sub-function.**
R12, R13, R14, R15, RBX, RDI, RSI, R8, R9, RDX	**See each TDG.VP.VMCALL sub-function. Register used in order.**
XMM0 – XMM15	If the corresponding bit in RCX is set to 1, the register value passed as-is from the host VMM's SEAMCALL (TDH.VP.ENTER) input. Otherwise, the register value is unmodified.

TDG.VP.VMCALL-Intel TDX paravirtualization sub-functions (specified in R11 when R10 is set to 0)

Table 2-3: TDG.VP.VMCALL codes

Sub-Function Number	Sub-Function Name
0x10000	GetTdVmCallInfo
0x10001	MapGPA
0x10002	GetQuote, *e.g.*, used for sending TDREPORT_STRUCT to VMM to request a TD Quote
0x10003	ReportFatalError
0x10004	SetupEventNotifyInterrupt
0x10005	Service

Table 2-4: TDG.VP.VMCALL-Instruction-execution sub-functions

Sub-Function Number Bits 15:0	Sub-Function Name
10	Instruction.CPUID
12	Instruction.HLT
30	Instruction.IO

Sub-Function Number Bits 15:0	Sub-Function Name
31	Instruction.RDMSR
32	Instruction.WRMSR
48	#VE.RequestMMIO
65	Instruction.PCONFIG

Note that some instructions that unconditionally cause #VE (such as WBINVD, MONITOR, MWAIT) do not have corresponding TDCALL [TDG.VP.VMCALL <Instruction>] leaves, since the TD has been designed with no deterministic way to confirm the result of those operations performed by the host VMM. In those cases, the goal is for the TD #VE handler to increment the RIP appropriately based on the VE information provided via TDCALL [TDG.VP.VEINFO.GET].

Completion-Status Codes

Table 2-5: TDCALL[TDG.VP.VMCALL]-Completion-Status Codes (Returned in RAX)

Completion-Status Code	Value	Description
TDX_SUCCESS	See Intel TDX-Architecture specification [3] for Function Completion Status Code.	TDCALL is successful
TDX_OPERAND_INVALID		Illegal leaf number
Other	See individual leaf functions	

Table 2-6: TDCALL[TDG.VP.VMCALL]- Sub-function Completion-Status Codes

(specified in R10 as output when R10 is set to 0 as input)

Completion-Status Code	Value	Description
TDG.VP.VMCALL_SUCCESS	0x0	TDCALL[TDG.VP.VMCALL] sub-function invocation was successful
TDG.VP.VMCALL_RETRY	0x1	TDCALL[TDG.VP.VMCALL] sub-function invocation must be retried
TDG.VP.VMCALL_OPERAND_INVALID	0x80000000 00000000	Invalid operand to TDG.VP.VMCALL sub-function
TDG.VP.VMCALL_GPA_INUSE	0x80000000 00000001	GPA already mapped
TDG.VP.VMCALL_ALIGN_ERROR	0x80000000 00000002	Operand (address) alignment error

3 TDG.VP.VMCALL Interface

From the perspective of the host VMM, TDCALL [TDG.VP.VMCALL] is a trap-like, VM exit into the host VMM reported via the SEAMRET instruction flow. By design, after the SEAMRET, the host VMM services the request specified in the parameters passed by the TD during the TDG.VP.VMCALL (that are passed via SEAMRET to the VMM) and resumes the TD via a SEAMCALL [TDH.VP.ENTER] invocation. Refer to the Intel TDX CPU Architecture specification [2] for details of the SEAMCALL and SEAMRET instructions. This chapter describes the designed sub-functions of the TDCALL [TDG.VP.VMCALL] interface between the TD and the VMM.

3.1 TDG.VP.VMCALL<GetTdVmCallInfo>

GetTdVmCallInfo TDG.VP.VMCALL is used to help request the host VMM enumerate which TDG.VP.VMCALLs are supported. This leaf is reserved for enumerating capabilities defined in this specification. VMMs may provide alternate, enumeration schemes using vendor-specific, TDG.VP.VMCALL namespace, as defined in 2.4.1.

Table 3-1: TDG.VP.VMCALL< GetTdVmCallInfo>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL< GetTdVmCallInfo> sub-function per Table 2-3.
R12	Leaf to enumerate TDG.VP.VMCALL functionality from this specification supported by the host. R12 must be set to 0, and successful execution of this TDG.VP.VMCALL is meant to indicate all TDG.VP.VMCALLs defined in this specification are supported by the host VMM. This register is reserved to extend TDG.VP.VMCALL enumeration in future versions.

Table 3-2: TDG.VP.VMCALL< GetTdVmCallInfo>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-instruction-return code.
R11	Leaf-specific output (when R12 is 0, will be returned as 0)
R12	Leaf-specific output (when R12 is 0, will be returned as 0)
R13	Leaf-specific output (when R12 is 0, will be returned as 0)
R14	Leaf-specific output (when R12 is 0, will be returned as 0)

Table 3-3: TDG.VP.VMCALL< GetTdVmCallInfo> Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful. The TD is free to use the GPA as a shared, memory page.

3.2 TDG.VP.VMCALL<MapGPA>

MapGPA TDG.VP.VMCALL is used to help request the host VMM to map a GPA range as private- or shared-memory mappings – this API may also be used to convert page mappings from private to shared. The GPA range passed in this operation can indicate if the mapping is requested for a shared or private memory – via the GPA.Shared bit in the start address. For example, to exchange data with the VMM, the TD may use this TDG.VP.VMCALL to request that a GPA range be mapped as a shared memory (for example, for paravirtualized IO) via the shared EPT. If the GPA (range) was already mapped as an active, private page, the host VMM may remove the private page from the TD by following the "Removing TD Private Pages" sequence in the Intel TDX-module specification [3] to safely block the mapping(s), flush the TLB and cache, and remove the mapping(s). The VMM is designed to be able to then map the specified GPA (range) in the shared-EPT structure and allow the TD to access the page(s) as a shared GPA (range).

If the Start GPA specified is a private GPA (GPA.S bit is clear), this MapGPA TDG.VP.VMCALL can be used to help request the host VMM map the specific, private page(s) (which mapping may involve converting the backing-physical page from a shared page to a private page). As intended in this case, the VMM

must unmap the GPA from the shared-EPT region and invalidate the TLB and caches for the TD vcpus to help ensure no stale mappings and cache contents exist. The aim is for the VMM to then follow the sequence specified in “Dynamically Adding TD Private Pages during TD Run Time” in the Intel TDX-module specification [3] to use TDH.MEM.PAGE.AUG to add the GPA(s) to the TD as pending, private mapping(s) in the secure-EPT. When the VMM responds to this TDG.VP.VMCALL with success, the goal is for the TD to execute TDCALL[TDG.MEM.PAGE.ACCEPT] to complete the process to make the page(s) usable as a private GPA inside the TD.

Upon MapGPA from shared to private, the VMM needs to check if the page is mapped by the IOMMU page table. If direct I/O is already enabled and the page is mapped, MapGPA should fail. This is equivalent to removing a page from a (legacy) guest with direct I/O enabled; the pages need to be pinned there. If the VMM provides a virtual IOMMU (vIOMMU) or cooperative IOMMU (coIOMMU), then the guest can indicate that it is not using that memory for DMA. In that case, MapGPA can: 1) Check that page is not pinned by vIOMMU; 2) Check that the page is not mapped in physical IOMMU. If 1 and 2 succeed, then unmap and remap it.

Table 3-4: TDG.VP.VMCALL<MapGPA>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<MapGPA> sub function per Table 2-3.
R12	4KB-aligned Start GPA of address range (Shared bit may be set or clear to indicate if a shared- or private-page mapping is desired) Shared-bit position is indicated by the GPA width [Guest-Physical-Address-Width-execution control is initialized by the host VMM for the TD during TDH.VP.INIT].
R13	Size of GPA region to be mapped (must be a multiple of 4KB)

Table 3-5: TDG.VP.VMCALL<MapGPA> Output Operands

Operand	Description
R10	TDG.VP.VMCALL-instruction-return code.
R11	GPA at which MapGPA failed (see failure or retry reason in TDG.VP.VMCALL specific status code.

Table 3-6: TDG.VP.VMCALL<MapGPA>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful. The TD is free to use the GPA (range) specified
TDG.VP.VMCALL_RETRY		TD must retry this operation for the pages in the region starting at the GPA specified in R11.
TDG.VP.VMCALL_INVALID_OPERAND		Invalid operand – for example, the GPA address is invalid (beyond GPAW).
TDG.VP.VMCALL_GPA_INUSE		GPA is already in use by the TD, for e.g., GPA used for hosting memory dedicated for IO. R11 specifies which GPA in the range specified was in use.
TDG.VP.VMCALL_ALIGN_ERROR		Alignment error for size or Start GPA.

3.3 TDG.VP.VMCALL<GetQuote>

GetQuote TDG.VP.VMCALL is a doorbell-like interface used to help send a message to the host VMM to queue operations that tend to be long-running operations. GetQuote is designed to invoke a request to generate a TD-Quote signing by a TD-Quoting Enclave operating in the host environment for a TD Report passed as a parameter by the TD. TDREPORT_STRUCT is a memory operand intended to be sent via the GetQuote TDG.VP.VMCALL to indicate the asynchronous service requested. For the GetQuote operation, the goal is the TDREPORT_STRUCT be received by the TD via a prior TDCALL[TDG.MR.REPORT]

in a 1024-byte buffer and placed in a shared-GPA space passed to the VMM as an operand in the GetQuote TDG.VP.VMCALL. In the case of this operation, the VMM can access the TDREPORT_STRUCT, queue the operation for a TD-Quoting enclave, and, when completed, return the Quote via the same, shared-memory area. For the TD to invoke the TDG.VP.VMCALL<GetQuote>, the host VMM can signal the event completion to the TD OS via a notification interrupt the host VMM injects into the TD (using the Event-notification vector registered via the SetupEventNotifyInterrupt TDG.VP.VMCALL).

Table 3-7: TDG.VP.VMCALL< GetQuote >-Input Operands

Operand	Description
R11	TDG.VP.VMCALL< GetQuote > sub-function per Table 2-3
R12	Shared 8KB GPA as input – the memory contains a TDREPORT_STRUCT. The same buffer is used as output – the memory contains a TD Quote.

Table 3-8: TDG.VP.VMCALL< GetQuote >-Output Operands

Operand	Description
R10	TDG.VP.VMCALL return code.

Table 3-9: TDG.VP.VMCALL< GetQuote >-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful received by the VMM. This status does not mean the TD Quote is generated or returned. The caller shall wait for event-notification to evaluate the output buffer to know if the TD Quote is generated successfully.
TDG.VP.VMCALL_RETRY		The TD should retry the operation
TDG.VP.VMCALL_INVALID_OPERAND		Invalid operand – for example, the GPA may be mapped as a private page.

3.4 TDG.VP.VMCALL<ReportFatalError>

The FatalError TDG.VP.VMCALL can inform the host VMM that the TD has experienced a fatal-error state and let the VMM access debug information. The output returned by the TDG.VP.VMCALL by the host VMM for Debug and Production versions of the platform may be different. This TDG.VP.VMCALL is intended to be used by the TD OS during early boot (in guest-firmware execution, for example) where some instructions like IN/OUT may be avoided to prevent causing a #VE; It may be also used by the TD guest post-boot when it detects an error (e.g., a security violation) and the TD wants to stop reliably

with information exposed to the host via the TD-specific error code (and additional information as a zero-terminated string via the shared memory 4KB region).

Table 3-10: TDG.VP.VMCALL< ReportFatalError >-Input Operands

Operand	Description		
R11	TDG.VP.VMCALL< ReportFatalError > sub-function per Table 2-3		
R12	**Bits**	**Name**	**Description**
	31:0	TD-specific error code	TD-specific error code Panic – 0x0 Values – 0x1 to 0xFFFFFFFF reserved
	62:32	TD-specific extended error code	TD-specific extended error code. TD software defined.
	63	GPA Valid	Set if the TD specified additional information in the GPA parameter (R13)
R13	4KB-aligned GPA where additional error data is shared by the TD. The VMM must validate that this GPA has the Shared bit set i.e., a shared-mapping is used, and that it is a valid mapping for the TD. This shared memory region is expected to hold a zero-terminated string. Shared-bit position is indicated by the GPA width [Guest-Physical-Address-Width-execution control is initialized by the host VMM for the TD during TDH.VP.INIT].		

Table 3-11: TDG.VP.VMCALL<ReportFatalError>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.

Table 3-12: TDG.VP.VMCALL< ReportFatalError >-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful

3.5 TDG.VP.VMCALL<SetupEventNotifyInterrupt>

The guest TD may request the host VMM specify which interrupt vector to use as an event-notify vector. This is designed as an untrusted operation; thus, the TD OS should be designed to not use the event notification for trusted operations. Example of an operation that can use the event notify is the host VMM signaling a device removal to the TD, in response to which a TD may unload a device driver.

The host VMM should use SEAMCALL [TDWRVPS] leaf to inject an interrupt at the requested-interrupt vector into the TD via the posted-interrupt descriptor. See Intel TDX-module specification [3] for TD-interrupt handling.

Table 3-13: TDG.VP.VMCALL< SetupEventNotifyInterrupt>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Setup Event Notify Interrupt> sub-function per Table 2-3
R12	Interrupt vector (valid values 32:255) selected by TD

Table 3-14: TDG.VP.VMCALL< SetupEventNotifyInterrupt >-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.

Table 3-15: TDG.VP.VMCALL< SetupEventNotifyInterrupt >-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		Invalid operand

3.6 TDG.VP.VMCALL<Instruction.CPUID>

Instruction.CPUID TDG.VP.VMCALL is designed to enable the TD-guest to request the VMM to emulate CPUID operation, especially for non-architectural, CPUID leaves.

Table 3-16: TDG.VP.VMCALL<Instruction.CPUID>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Instruction.CPUID>-Instruction-execution sub-functions per Table 2-3
R12	EAX
R13	ECX

Table 3-17: TDG.VP.VMCALL<Instruction.CPUID>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.
R12	EAX
R13	EBX
R14	ECX
R15	EDX

Table 3-18: TDG.VP.VMCALL<Instruction.CPUID>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		Invalid CPUID requested

3.7 TDG.VP.VMCALL<#VE.RequestMMIO>

This TDG.VP.VMCALL is used to help request the VMM perform emulated-MMIO-access operation. The VMM may emulate MMIO space in shared-GPA space. The VMM can induce a #VE on these shared-GPA accesses by mapping shared GPAs with the suppress-VE bit cleared in the EPT Entries corresponding to these mappings. In response to the #VE, the TD can use the TDCALL[TDG.VP.VEINFO.GET] to get the Virtualization-Exception-Information Fields (See **Error! Reference source not found.**) and validate that t he #VE exit reason is 48 (EPT violation causing #VE). After the TD software decodes the instruction causing the #VE locally and validating the accessed region and source of access, the TD may choose to use this TDG.VP.VMCALL to request MMIO read/write operations. The VMM may emulate the access based on the inputs provided by the TD. However, note that, like other TDG.VP.VMCALLs, this TDCALL is designed as an untrusted operation and to be used for untrusted IO with other cryptographic protection for the TD data provided by the TD itself.

Table 3-19: TDG.VP.VMCALL<RequestMMIO>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<RequestMMIO> sub-function per Table 2-3
R12	Size of access. 1=1byte, 2=2bytes, 4=4bytes, 8=8bytes. All rest value = reserved.
R13	Direction. 0=Read, 1=Write. All rest value = reserved.
R14	MMIO Address
R15	Data to write, if R13 is 1.

Table 3-20: TDG.VP.VMCALL<Instruction.MMIO>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.
R11	Data to read, if R13 is 0.

Table 3-21: TDG.VP.VMCALL<Instruction.MMIO>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		If invalid operands provided by the TD, *e.g.*, MMIO address

3.8 TDG.VP.VMCALL<Instruction.HLT>

Instruction.HLT TDG.VP.VMCALL is used to help perform HLT operation. The TD guest informs the VMM regarding the TD's interrupt (blocked) status via this interface.

Table 3-22: TDG.VP.VMCALL<Instruction.HLT>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Instruction.HLT>-Instruction-execution sub-functions per Table 2-3
R12	Interrupt Blocked Flag. The TD is expected to clear this flag iff RFLAGS.IF == 1 or the TDCALL instruction (that invoked TDG.VP.TDVMCALL(Instruction.HLT)) immediately follows an STI instruction, otherwise this flag should be set.

Table 3-23: TDG.VP.VMCALL<Instruction.HLT>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.

Table 3-24: TDG.VP.VMCALL<Instruction.HLT>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful

3.9 TDG.VP.VMCALL<Instruction.IO>

Instruction.IO TDG.VP.VMCALL is used to help request the VMM perform IO operations.

Table 2-25: TDG.VP.VMCALL<Instruction.IO>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Instruction.IO>-Instruction-execution sub-functions per Table 2-3
R12	Size of access. 1=1byte, 2=2bytes, 4=4bytes. All rest value = reserved.
R13	Direction. 0=Read, 1=Write. All rest value = reserved.
R14	Port number
R15	Data to write, if R13 is 1.

Table 2-26: TDG.VP.VMCALL<Instruction.IO>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.
R11	Data to read, if R13 is 0.

Table 2-27: TDG.VP.VMCALL<Instruction.IO>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		Invalid-IO-Port access

3.10 TDG.VP.VMCALL<Instruction.RDMSR>

Instruction.RDMSR TDG.VP.VMCALL is used to hep perform RDMSR operation.

Table 2-28: TDG.VP.VMCALL<Instruction.RDMSR>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Instruction.RDMSR> Instruction execution sub-functions per Table 2-3
R12	MSR Index

Table 2-29: TDG.VP.VMCALL<Instruction.RDMSR>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.
R11	MSR Value

Table 2-30: TDG.VP.VMCALL<Instruction.RDMSR>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		Invalid MSR rd/wr requested or access-denied

3.11 TDG.VP.VMCALL<Instruction.WRMSR>

Instruction.WRMSR TDG.VP.VMCALL is used to help perform WRMSR operation.

Table 2-31: TDG.VP.VMCALL<Instruction.WRMSR>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Instruction.WRMSR>-Instruction-execution sub-functions per Table 2-3
R12	MSR Index
R13	MSR Value

Table 2-32: TDG.VP.VMCALL<Instruction.WRMSR>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.

Table 2-33: TDG.VP.VMCALL<Instruction.WRMSR>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		Invalid MSR rd/wr requested or access-denied

3.12 TDG.VP.VMCALL<Instruction.PCONFIG>

Instruction.VMCALL PCONFIG is used to help perform Instruction-PCONFIG operation.

Table 2-34: TDG.VP.VMCALL<Instruction.PCONFIG>-Input Operands

Operand	Description
R11	TDG.VP.VMCALL<Instruction.PCONFIG> sub-function per Table 2-3
R12	PCONFIG-Leaf function requested
R13, R14, R15	Leaf-specific purpose (See PCONFIG ISA definition in MKTME spec. [4]

Table 2-35: TDG.VP.VMCALL<Instruction.PCONFIG>-Output Operands

Operand	Description
R10	TDG.VP.VMCALL-return code.
R11	VMM-Vendor Specific
R12, R13, R14, R15, RBX, RDI, RSI, R8, R9, RDX	VMM-Vendor Specific
XMM0 – XMM15	If RCX bit 1 is set, the XMM content is set by VMM host when executing SEAMCALL(TDENTER). Otherwise, the XMM content is unmodified.

Table 2-36: TDG.VP.VMCALL<Instruction.PCONFIG>-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		If PCONFIG-operation requested is invalid

3.13 TDG.VP.VMCALL <Service>

In Service TD scenario, there is need to define interfaces for the command/response that may have long latency, such as communicating with local device via Secure Protocol and Data Model (SPDM), communicating with remote platform via Transport Layer Security (TLS) Protocol, or communicating with a Quoting Enclave (QE) on attestation or mutual authentication.

There is also need that the VMM may notify a service TD to do some actions, such as Migration TD (MigTD).

We define Command/Response Buffer (CRB) DMA interface.

Table 3-37: TDG.VP.VMCALL< Service >-Input Operands

Operand	Description
R11	TDG.VP.VMCALL< Service > sub-function per **Error! Reference source not found.**
R12	Shared 4KB aligned GPA as input – the memory contains a Command. It could be more than one 4K pages.
R13	Shared 4KB aligned GPA as output – the memory contains a Response. It could be more than one 4K pages.
R14	Event notification interrupt vector - (valid values 32~255) selected by TD 0: blocking action. VMM need get response then return. 1~31: Reserved. Should not be used. 32~255: Non-block action. VMM can return immediately and signal the interrupt vector when the response is ready.
R15	Timeout– Maximum wait time for the command and response. 0 means infinite wait.

Table 3-38: TDG.VP.VMCALL< Service >-Output Operands

Operand	Description
R10	TDG.VP.VMCALL return code.

Table 3-39: TDG.VP.VMCALL< Service >-Status Codes

Error Code	Value	Description
TDG.VP.VMCALL_SUCCESS	See values in Table 2-6	TDG.VP.VMCALL is successful
TDG.VP.VMCALL_INVALID_OPERAND		Invalid operand – for example, the GPA may be mapped as a private page. Or the interrupt vector is invalid.

Table 3-40: TDG.VP.VMCALL< Service >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
GUID	0	16	A unique GUID to identify the service. This field is always filled by TD.
Length	16	4	Size in bytes of the command buffer, including the GUID and Length. (24 + N) This field is always filled by TD.
Reserved	20	4	Reserved
Data	24	N	GUID specific command data. This field is always filled by TD.

Table 3-41: TDG.VP.VMCALL< Service >-response buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
GUID	0	16	A unique GUID to identify the service. This field is always filled by TD.
Length	16	4	Size in bytes of the response buffer, including the GUID and Length. (24 + N) When TD uses the VMCALL, this field is filled by TD to indicate the maximum allocated response buffer size. When VMM triggers interrupt vector, this field is filled by VMM to indicate the returned response buffer size.
Status	20	24	Common Status Code for response. 0: Command is sent, and response is returned. 1: Device error 2: Timeout 3: Response buffer too small 4: Bad command buffer size 5: Bad response buffer size 6: Service busy 7: Invalid Parameter 8: Out of resource 0xFFFFFFFE: Unsupported. 0xFFFFFFFF: Reserved – should be filled by TD, so that TD can check if the response is returned by VMM. This field is always filled by VMM. (*)
Data	24	N	GUID specific response data. This field is always filled by VMM.

(*) *NOTE: Status field only means the VMM sends the command receives a response. It does not mean the response contain a success code. For example, an SPDM responder may return an SPDM error in a SPDM response message. The VMM will fill SUCCESS in this status. The SPDM requester in TD should parse the SPDM response message to get the SPDM error code.*

The detail flow is:

Step 1, the TD will:

1.1) Prepare an event notification interrupt vector.

1.2) Prepare a shared command buffer and put command there, including GUID, length and data.

1.3) Prepare a shared response buffer and put the GUID and maximum length of response buffer there.

1.4) Trigger TDVMCALL.

Step 2, the VMM will:

2.1) Setup the context for this TDVMCALL <Service> (For example, save the response buffer and event notification interrupt vector)

2.2) Send command

2.3) Prepare for response (Interrupt or Poll)

2.2) return TDVMCALL to TD.

Step 3, the TD will:

3.1) Free the shared command buffer and make it private. (Optional, if the TD wants to reuse the buffer later)

Step 4, once VMM gets the response, the VMM will

4.1) Check if the Response.Length is large enough to hold the response.

4.2) Fill the Response.Data field with response.

4.3) Free the context for this TDVMCALL <Service>

4.3) Trigger Event Notification Interrupt to TD.

Step 5, once TD gets the event notification, the TD will

5.1) Read the data from the response to private.

5.2) Free the shared response buffer and make it private. (Optional, if the TD wants to reuse the buffer later)

5.3) If the returned data includes length, offset, index, etc, apply the side channel mitigation, such as lfence(), before parse the data.

5.4) Parse the response in private memory. NOTE: This is from untrusted source. The TD must check the data before use it.

3.13.1 TDG.VP.VMCALL <Service.Query>

This generic service is to allow TD to query the capability.

```
// {FB6FC5E1-3378-4ACB-8964-FA5EE43B9C8A}
#define VMCALL_SERVICE_COMMON_GUID \
{0xfb6fc5e1, 0x3378, 0x4acb, 0x89, 0x64, 0xfa, 0x5e, 0xe4, 0x3b, 0x9c, 0x8a}
```

Table 3-42: TDG.VP.VMCALL< Service.Query >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	0: **Query**
Reserved	2	2	Reserved
GUID	4	16	The Service GUID to query

Table 3-43: TDG.VP.VMCALL< Service.Query >-response buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	0: **Query**
Status	2	1	0: Service supported 1: Service unsupported
Reserved	3	1	Reserved
GUID	4	16	The Service GUID to query

3.13.2 TDG.VP.VMCALL <Service.MigTD>

This is used to allow MigTD to get the migration information from VMM.

```
#define VMCALL_SERVICE_MIGTD_GUID \
{0xe60e6330, 0x1e09, 0x4387, 0xa4, 0x44, 0x8f, 0x32, 0xb8, 0xd6, 0x11, 0xe5}
```

3.13.2.1 TDG.VP.VMCALL <Service.MigTD.Shutdown>

This is used to allow a service TD to shutdown itself after it finishes the task.

Table 3-44: TDG.VP.VMCALL< Service.MigTD.Shutdown >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	0: **Shutdown**
Reserved	2	2	Reserved

No response data is required.

3.13.2.2 TDG.VP.VMCALL <Service.MigTD.WaitForRequest>

Table 3-45: TDG.VP.VMCALL< Service.MigTD.WaitForRequest >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	1: **Wait for request**
Reserved	2	2	Reserved

Table 3-46: TDG.VP.VMCALL< Service.MigTD.WaitForRequest >-response buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	1: **Wait for request**
Operation	2	1	0: No-op **1: Start Migration** 2~0xFF: reserved This field is used to perform corresponding migration command.
Reserved	3	1	Reserved
Migration Information	4	Variable	Migration related GUID extension HOB, including **MIGTD_MIGRATION_INFORMATION** HOB and others.

3.13.2.2.1 Migration Request Information GUID Extension HOB

This GUID extension HOB reports the migration request information.

```
#define MIGTD_MIGRATION_INFORMATION_HOB_GUID \
{0x42b5e398, 0xa199, 0x4d30, 0xbe, 0xfc, 0xc7, 0x5a, 0xc3, 0xda, 0x5d, 0x7c}

typedef struct {
  UINT64              MigRequestID;
  BOOLEAN             MigrationSource;
  UINT8               TargetTD_UUID[32];
  UINT64              BindingHandle;
  UINT64              MigPolicyID;
  UINT64              CommunicationID;
} MIGTD_MIGRATION_INFORMATION;
```

MigRequestID	The ID for the migration request. It is used in **TDG.VP.VMCALL <Service.MigTD.ReportStatus>** in **TDG.VP.VMCALL <Service.MigTD.Send>** and **TDG.VP.VMCALL <Service.MigTD.Receive>**.
MigrationSource	TRUE: This MigTD is MigTD-s; FALSE: This MigTD is MigTD-d.
TargetTD_UUID	The UUID for the target TD returned from **SEAMCALL[TDH.SERVTD.BIND]**.

BindingHandle	The BindingHandle for the MigTD and the target TD returned from **SEAMCALL[TDH.SERVTD.BIND]**.
MigPolicyID	The ID for the migration policy.
CommunicationID	The ID for the VMM communication channel.

3.13.2.2.2 Stream Socket Info GUID Extension HOB

This GUID extension HOB reports the VMCALL based stream socket information.

```
#define MIGTD_STREAM_SOCKET_INFO_HOB_GUID \
{0x7a103b9d, 0x552b, 0x485f, 0xbb, 0x4c, 0x2f, 0x3d, 0x2e, 0x8b, 0x1e, 0xe}

typedef struct {
  UINT64                CommunicationID;
  UINT64                MigTdCid;
  UINT32                MigChannelPort;
  UINT32                QuoteServicePort;
} MIGTD_STREAM_SOCKET_INFO;
```

CommunicationID	A unique identifier for this communication. It can be used in **MIGTD_MIGRATION_INFORMATION** HOB.
MigTdCid	The context ID (CID) for the MigTD.
MigChannelPort	The listening port of the MigTD or VMM for the migration secure communication channel.
QuoteServicePort	The listening port of the VMM for the quote service channel.

3.13.2.2.3 Runtime Migration Policy GUID Extension HOB

This GUID extension HOB reports the runtime migration policy.

```
#define MIGTD_MIGPOLICY_HOB_GUID \
{0xd64f771a, 0xf0c9, 0x4d33, 0x99, 0x8b, 0xe, 0x3d, 0x8b, 0x94, 0xa, 0x61}

typedef struct {
  UINT64                MigPolicyID;
  UINT32                MigPolicySize;
  UINT8                 MigPolicy[];
```

} MIGTD_MIGPOLICY_INFO;

MigPolicyID	A unique identifier for this policy. It can be used in **MIGTD_MIGRATION_INFORMATION** HOB.
MigPolicySize	The size in bytes of the migration policy.
MigPolicy	The migration policy data.

3.13.2.3 TDG.VP.VMCALL <Service.MigTD.ReportStatus>

Table 3-47: TDG.VP.VMCALL< Service.MigTD.ReportStatus >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	2: **Report Status**
Operation	2	1	Same as the "Operation" in **TDG.VP.VMCALL <Service.MigTD.WaitForRequest>**
Status	3	1	0: SUCCESS 1: INVALID_PARAMETER 2: UNSUPPORTED 3: OUT_OF_RESOURCE 4: TDX_MODULE_ERROR 5: NETWORK_ERROR 6: SECURE_SESSION_ERROR 7: MUTUAL_ATTESTATION_ERROR 8: MIGPOLICY_ERROR 0xFF: MIGTD_INTERNAL_ERROR 0x0A~0xFE: Reserved
MigRequestID	4	8	The MigRequestID in **MIGTD_MIGRATION_INFORMATION** HOB

Table 3-48: TDG.VP.VMCALL< Service.MigTD.ReportStatus >-response buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	2: **Report Status**
Reserved	2	2	Reserved

3.13.2.4 TDG.VP.VMCALL <Service.MigTD.Send>

The two MigTDs use this VMCALL and next VMCALL to transmit and receive the communication packet to each other via the VMM.

The communication packet is transparent to the VMM. The VMM shall treat it as binary blob and do not parse it. The VMM shall guarantee the connection is reliable. For example, no communication packet will be lost. No communication packet will arrive out of order.

The MigTD shall guarantee the communication request for one MigRequestID is sequential. Multiple requests for one MigRequestID at same time are not allowed. If the VMM receives another request while it is waiting for the response of the previous request with the same MigRequestID, the VMM shall return **Service busy** error status. The MigTD may send multiple requests for different MigRequestID at same time, the VMM shall support this case and server all requests. If the VMM may run out of resource to create a new context to communicate with the remote, the VMM shall return **Out of resource** error status.

If the VMM has hardware error and fails to send the command or receive the response, the VMM shall return **Device error** status. If the VMM fails to send the command or receive the response in timeout period indicated by the MigTD, the VMM shall return **Timeout** status.

The MigTD shall provide a big enough response buffer. If the response buffer is too small to hold the response from the peer MigTD, the VMM shall return **Receive buffer too small** error status.

The MigTD shall provide a reasonable size command buffer and response buffer to allow VMM allocate the memory to handle the input or output. The VMM shall supports **at least 64KB** command buffer and response buffer. If the MigTD request an unreasonable large command buffer or response buffer, such as 1GB, the VMM may return **Bad command buffer size** or **Bad response buffer size** error status.

The MigTD may input zero command buffer size, when the MigTD-d wants to wait for the first request message from MigTD-s. The MigTD may input zero response buffer size, when a MigTD sends the last handshake message to the peer. The VMM shall handle those cases correctly. If the MigTD inputs zero command buffer size and zero response buffer size at same time, the VMM shall return **Invalid parameter** error status.

For the detail of stream communication flow, please refer to VMCALL stream message section.

Table 3-48: TDG.VP.VMCALL< Service.MigTD.Send >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	3: **Send**
Reserved	2	2	Reserved
MigRequestID	4	8	The MigRequestID in **MIGTD_MIGRATION_INFORMATION** HOB
Stream Message Header	12	32	The stream message header. See VMCALL stream message section - **VMCALL_STREAM_MESSAGE_HEADER**.
MigTD communication packet	44	N	This field is only present when **VMCALL_STREAM_MESSAGE_HEADER.Operation** is **VMCALL_STREAM_OP_RW**. Size in bytes of the MigTD communication packet to be sent. The format of the packet is defined by the MigTD. The VMM shall send the binary blob to the peer MigTD.

Table 3-49: TDG.VP.VMCALL< Service.MigTD.Send >-response buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	3: **Send**
Reserved	2	2	Reserved
MigRequestID	4	8	The MigRequestID in **MIGTD_MIGRATION_INFORMATION** HOB

3.13.2.1 TDG.VP.VMCALL <Service.MigTD.Receive>

Table 3-50: TDG.VP.VMCALL< Service.MigTD.Receive >-command buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	4: **Receive**
Reserved	2	2	Reserved
MigRequestID	4	8	The MigRequestID in **MIGTD_MIGRATION_INFORMATION** HOB

Table 3-51: TDG.VP.VMCALL< Service.MigTD.Receive >-response buffer layout

Field	Offset (Bytes)	Length (bytes)	Description
Version	0	1	0: for this data structure
Command	1	1	4: **Receive**
Reserved	2	2	Reserved
MigRequestID	4	8	The MigRequestID in **MIGTD_MIGRATION_INFORMATION** HOB
Stream Message Header	12	32	The stream message header. See VMCALL stream message section - **VMCALL_STREAM_MESSAGE_HEADER**.
MigTD communication packet	44	N	This field is only present when **VMCALL_STREAM_MESSAGE_HEADER.Operation** is **VMCALL_STREAM_OP_RW**. Size in bytes of the MigTD communication packet being received. The format of the packet is defined by the MigTD. The VMM shall receive the binary blob from the peer MigTD.

3.13.2.1.1 Stream Socket Message

A service TD may setup a stream connection with a VMM. Either a service TD or a VMM can be the client or the server. Every stream message starts with a **VMCALL_STREAM_MESSAGE_HEADER**, followed by the payload if the operation is **VMCALL_STREAM_OP_RW**.

In order to create a stream connection, the client sends **VMCALL_STREAM_OP_REQUEST** packet. If the server is listening, then it sends **VMCALL_STREAM_OP_RESPONSE** packet and the stream connection is established. If the server is not listening or has no resource, then it sends **VMCALL_STREAM_OP_RESET** packet.

Once the stream connection is established, both entities may send **VMCALL_STREAM_OP_RW** packet to the peer.

When an entity does not need send more data or receive more data, it sends **VMCALL_STREAM_OP_SHUTDOWN** packet. The peer sends **VMCALL_STREAM_OP_RESET** response to confirm. The stream connection is terminated. Even if the response is not received, the connection is still terminated with a timeout period.

```
typedef struct {
  UINT64                SourceCid;
  UINT64                DestinationCid;
  UINT32                SourcePort;
  UINT32                DestinationPort;
  UINT32                Length;
  UINT16                Operation;
  UINT16                Reserved;
} VMCALL_STREAM_MESSAGE_HEADER;
```

SourceCid	The context ID of the source.
DestinationCid	The context ID of the destination.
SourcePort	The port of the source.
DestinationPort	The port of the destination.
Length	The length of the payload. It does not include this header.
Operation	The message operation. See **VMCALL_STREAM_MESSAGE_OPERATION**.

```
typedef enum {
  VMCALL_STREAM_OP_RESERVED = 0,
  // stream connection
  VMCALL_STREAM_OP_REQUEST = 1,
  VMCALL_STREAM_OP_RESPONSE = 2,
  VMCALL_STREAM_OP_RESET = 3,
  VMCALL_STREAM_OP_SHUTDOWN = 4,
  // send payload message
  VMCALL_STREAM_OP_RW = 5,
} VMCALL_STREAM_MESSAGE_OPERATION;
```

The VMM CID is always set to 0x2. The MigTD CID is passed from VMM dynamically. The server listening port of VMM or MigTD can be predefined or passed from VMM dynamically.

When the MigTD acts as a client and the VMM acts as a server. Below communication flow is used:

MigTD: {MigTdCid, VmmCid, **MigTdPort**, **VmmListeningPort**, OP_REQUEST}

VMM: {VmmCid, MigTdCid, VmmListeningPort, MigTdPort, OP_RESPONSE}

MigTD: {MigTdCid, VmmCid, MigTdPort, VmmListeningPort, OP_RW, payload}

VMM: {VmmCid, MigTdCid, VmmListeningPort, MigTdPort, OP_RW, payload}

……

MigTD: {MigTdCid, VmmCid, MigTdPort, VmmListeningPort, OP_SHUTDOWN}

VMM: {VmmCid, MigTdCid, VmmListeningPort, MigTdPort, OP_RESET}

When the MigTD acts as a server and the VMM acts as a client. Below communication flow is used:

VMM: {VmmCid, MigTdCid, **VmmPort**, **MigTdListeningPort**, OP_REQUEST}

MigTD: {MigTdCid, VmmCid, MigTdListeningPort, VmmPort, OP_RESPONSE}

VMM: {VmmCid, MigTdCid, VmmPort, MigTdListeningPort, OP_RW, payload}

MigTD: {MigTdCid, VmmCid, MigTdListeningPort, VmmPort, OP_RW, payload}

……

VMM: {VmmCid, MigTdCid, VmmPort, MigTdListeningPort, OP_SHUTDOWN}

MigTD: {MigTdCid, VmmCid, MigTdListeningPort, VmmPort, OP_RESET}

4 TD-Guest-Firmware Interfaces

4.1 ACPI MADT Multiprocessor Wakeup Table

The guest firmware is designed to publish a multiprocessor-wakeup structure to let the guest-bootstrap processor wake up guest-application processors with a mailbox. The mailbox is memory that the guest firmware can reserve so each guest, virtual processor can have the guest OS send a message to them.

For detailed information about the Multiprocessor Wakeup Table, please refer to ACPI 6.4 specification.

4.2 Memory Map

The memory in the TD guest-environment can be:

1) Private memory – SEAMCALL[TDH.MEM.PAGE.ADD] by VMM or TDCALL [TDG.MEM.PAGE.ACCEPT] by TDVF with S-bit clear in page table.

2) Shared memory – SEAMCALL[TDH.MEM.PAGE.ADD] by VMM or TDCALL [TDG.MEM.PAGE.ACCEPT] by TDVF with S-bit set in page table.

3) Unaccepted memory – SEAMCALL[TDH.MEM.PAGE.AUG] by VMM and not accepted by TDVF yet.

4) Memory-mapped IO (MMIO) - Shared memory accessed via TDVF via TDVMCALL<#VE.RequestMMIO>.

If a TD-memory region is private memory, the TD owner shall have the final UEFI-memory map report the region with **`EfiReservedMemoryType`**, **`EfiLoaderCode`**, **`EfiLoaderData`**, **`EfiBootServiceCode`**, **`EfiBootServiceData`**, **`EfiRuntimeServiceCode`**, **`EfiRuntimeServiceData`**, **`EfiConventionalMemory`**, **`EfiACPIReclaimMemory`**, **`EfiACPIMemoryNVS`**.

If a TD-memory region is shared memory, the TD owner shall convert it to private memory before transfer to OS kernel.

If a TD-memory region is unaccepted memory and requires TDCALL [TDG.MEM.PAGE.ACCEPT] in the TD guest OS, then the TD owner shall have the final UEFI-memory map report this region with **`EfiUnacceptedMemoryType`**. Please refer to UEFI 2.9 specification.

If a memory region is MMIO, it is designed to only be accessed via TDVMCALL<#VE.RequestMMIO> and not via direct memory read or write. Accordingly, as designed, there is no need to report this region in UEFI-memory map, because no RUNTIME attribute is required. The full, MMIO regions is designed to be reported in ACPI ASL code via memory-resource descriptors.

Table 4-1: TDVF-memory map for OS

UEFI Memory Type	Usage	TD-Memory Type	OS Action
EfiReservedMemoryTpe	Firmware-Reserved region, such as flash.	Private	Reserved.
EfiLoaderCode	UEFI-Loader Code	Private	Use after EBS.
EfiLoaderData	UEFI-Loader Data	Private	Use after EBS.
EfiBootServicesCode	UEFI-Boot-Service Code	Private	Use after EBS.
EfiBootServicesData	UEFI-Boot-Service Data	Private	Use after EBS.
EfiRuntimeServicesCode	UEFI-Runtime-Service Code	Private	Map-virtual address. Reserved.
EfiRuntimeServicesData	UEFI-Runtime-Service Data	Private	Map-virtual address. Reserved.
EfiConventionalMemory	Freed memory (Private)	Private	Use directly.
EfiACPIReclaimMemory	ACPI table.	Private	Use after copy ACPI table.
EfiACPIMemoryNVS	Firmware Reserved for ACPI, such as the memory used in ACPI OpRegion	Private	Reserved.
EfiMemoryMappedIO	No need to report the MMIO region, as no RUNTIME-virtual address is required for TD. The full MMIO should be reported in ACPI-ASL code.	N/A	N/A
EfiUnacceptedMemoryType	UEFI-Boot-Service Data (Shared Memory) VMM-shared buffer.	Unaccepted	Use after EBS and converting to private page. ========== TDCALL[TDG.VP.VMCALL] <MapGPA> TDCALL[TDG.MEM.PAGE.ACCEPT]

For non-UEFI system, the memory map can be reported via E820 table.

If a TD-memory region is private memory, the TD Shim shall have the final memory map report the region with **AddressRangeMemory**, **AddressRangeReserved**, **AddressRangeACPI**, or **AddressRangeNVS**.

If a TD-memory region is shared memory, the TD Shim shall convert it to private memory before transfer to OS kernel.

If a TD-memory region is unaccepted memory and requires TDCALL [TDG.MEM.PAGE.ACCEPT] in the TD guest OS, then the TD Shim shall have the final memory map report this region with **AddressRangeUnaccepted**.

If a memory region is MMIO, it is designed to only be accessed via TDVMCALL<#VE.RequestMMIO> and not via direct memory read or write. Accordingly, as designed, there is no need to report this region in the final memory map.

Table 4-2: TDVF E820 memory map for OS

E820 Memory Type	Usage	TD-Memory Type	OS Action
AddressRangeMemory	Usable by OS.	Private	Use directly
AddressRangeReserved	Firmware-Reserved region, such as flash.	Private	Reserved.
AddressRangeACPI	ACPI table.	Private	Use after copy ACPI table.
AddressRangeNVS	Firmware Reserved for ACPI, such as the memory used in ACPI OpRegion	Private	Reserved.
AddressRangeUnaccepted	Allocated by VMM, but not accepted by TD guest yet.	Unaccepted	Use after convert to private page.

4.3 TD Measurement

4.3.1 TCG-Platform-Event Log

If TD-Guest Firmware supports measurement and an event is created, TD-Guest Firmware is designed to report the event log with the same data structure in TCG-Platform-Firmware-Profile specification with **EFI_TCG2_EVENT_LOG_FORMAT_TCG_2** format.

The index created by the TD-Guest Firmware in the event log should be the index for the TD-measurement register.

Table 4-3: TD-Event-Log-PCR-Index Interpretation

TD-Register Index	TDX-measurement register
0	MRTD
1	RTMR[0]
2	RTMR[1]
3	RTMR[2]
4	RTMR[3]

4.3.2 EFI_TD_PROTOCOL

If TD-Guest Firmware supports measurement, the TD Guest Firmware is designed to produce EFI_TD_PROTOCOL with new GUID **EFI_TD_PROTOCOL_GUID** to report event log and provide hash capability.

EFI_TD_PROTOCOL

Summary

This protocol abstracts the TD measurement operation in UEFI guest environment.

GUID

```
#define EFI_TD_PROTOCOL_GUID \
{0x96751a3d, 0x72f4, 0x41a6, {0xa7, 0x94, 0xed, 0x5d, 0xe, 0x67, 0xae,
0x6b }}
```

Protocol Interface Structure

```
typedef struct _EFI_TD_PROTOCOL {
  EFI_TD_GET_CAPABILITY          GetCapability;
  EFI_TD_GET_EVENT_LOG           GetEventLog;
  EFI_TD_HASH_LOG_EXTEND_EVENT   HashLogExtendEvent;
  EFI_TD_MAP_PCR_TO_MR_INDEX     MapPcrToMrIndex;
} EFI_TD_PROTOCOL;
```

Parameters

GetCapability

Provide protocol capability information and state information. See the **GetCapability()** function description.

GetEventLog

Allow a caller to retrieve the address of a given event log and its last entry. See the **GetEventLog()** function description.

HashLogExtendEvent

Provide callers with an opportunity to extend and optionally log events without requiring knowledge of actual TD command. See the **HashLogExtendEvent()** function description.

MapPcrToMrIndex

Provide callers information on TPM PCR to TDX measurement register (MR) mapping. See the **MapPcrToMrIndex()** function description.

Description

The **EFI_TD_PROTOCOL** is used to abstract the TDX measurement related action in TDX UEFI guest environment.

EFI_TD_PROTOCOL.GetCapability

Summary

This service provides protocol capability information and state information.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TD_GET_CAPABILITY)(
  IN EFI_TD_PROTOCOL                      *This,
  IN OUT EFI_TD_BOOT_SERVICE_CAPABILITY *ProtocolCapability
  );
```

Parameters

This

The protocol interface pointer.

ProtocolCapability

The caller allocates memory for a EFI_TD_BOOT_SERVICE_CAPABILITY structure and sets the size field to the size of the structure allocated. The callee fills in the fields with the EFI protocol capability information and the current EFI TD state information up to the number of fields which fit within the size of the structure passed in.

Description

This function provides protocol capability information and state information.

Status Code Returned

EFI_SUCCESS	Operation completed successfully.
EFI_INVALID_PARAMETER	One or more of the parameters are incorrect. The *ProtocolCapability* variable will not be populated.
EFI_DEVICE_ERROR	The command was unsuccessful. The *ProtocolCapability* variable will not be populated.
EFI_BUFFER_TOO_SMALL	The *ProtocolCapability* variable is too small to hold the full response. It will be partially populated (required Size field will be set).

Related Definitions

```
typedef struct {
  //
  // Allocated size of the structure
  //
  UINT8                             Size;
  //
  // Version of the EFI_TD_BOOT_SERVICE_CAPABILITY structure.
  // For this version of the protocol,
  // the Major version shall be set to 1
  // and the Minor version shall be set to 1.
  //
  EFI_TD_VERSION                    StructureVersion;
  //
  // Version of the EFI TD protocol.
  // For this version of the protocol,
  // the Major version shall be set to 1
  // and the Minor version shall be set to 1.
  //
  EFI_TD_VERSION                    ProtocolVersion;
  //
  // Supported hash algorithms
  //
  EFI_TD_EVENT_ALGORITHM_BITMAP     HashAlgorithmBitmap;
  //
  // Bitmap of supported event log formats
  //
  EFI_TD_EVENT_LOG_BITMAP           SupportedEventLogs;

  //
  // False = TD not present
  //
  BOOLEAN                           TdPresentFlag;
} EFI_TD_BOOT_SERVICE_CAPABILITY;

typedef struct {
  UINT8 Major;
  UINT8 Minor;
```

```
} EFI_TD_VERSION;

typedef UINT32      EFI_TD_EVENT_LOG_BITMAP;
typedef UINT32      EFI_TD_EVENT_ALGORITHM_BITMAP;

#define EFI_TD_BOOT_HASH_ALG_SHA384     0x00000004
```

EFI_TD_PROTOCOL.GetEventLog

Summary

This service allows a caller to retrieve the address of a given event log and its last entry.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TD_GET_EVENT_LOG)(
  IN EFI_TD_PROTOCOL                    *This,
  IN EFI_TD_EVENT_LOG_FORMAT            EventLogFormat,
  OUT EFI_PHYSICAL_ADDRESS              *EventLogLocation,
  OUT EFI_PHYSICAL_ADDRESS              *EventLogLastEntry,
  OUT BOOLEAN                           *EventLogTruncated
  );
```

Parameters

This

The protocol interface pointer.

EventLogFormat

The type of event log for which the information is requested.

EventLogLocation

A pointer to the memory address of the event log.

EventLogLastEntry

If the event log contains more than one entry, this is a pointer to the address of the start of the last entry in the event log in memory.

EventLogTruncated

If the event log is missing at least one entry because one event would have exceeded the area allocated for the event, this value is set to TRUE. Otherwise, this value will be FALSE and the event log is complete.

Description

This function allows a caller to retrieve the address of a given event log and its last entry.

Status Code Returned

EFI_SUCCESS	Operation completed successfully.
EFI_INVALID_PARAMETER	One or more of the parameters are incorrect.

Related Definitions

```
typedef UINT32                              EFI_TD_EVENT_LOG_FORMAT;

#define EFI_TD_EVENT_LOG_FORMAT_TCG_2       0x00000002
```

EFI_TD_PROTOCOL.HashLogExtendEvent

Summary

This service provides callers with an opportunity to extend and optionally log events without requiring knowledge of actual TD command.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TD_GET_EVENT_LOG)(
  IN EFI_TD_PROTOCOL                        *This,
  IN UINT64                                 Flags,
  IN EFI_PHYSICAL_ADDRESS                   DataToHash,
  IN UINT64                                 DataToHashLen,
  IN EFI_TD_EVENT                           *EfiTdEvent
  );
```

Parameters

This

The protocol interface pointer.

Flags

Bitmap providing additional information.

DataToHash

Physical address of the start of the data buffer to be hashed.

DataToHashLen

The length in bytes of the buffer referenced by *DataToHash*.

EfiTdEvent

Pointer to the data buffer containing information about the event.

Description

This function provides callers with an opportunity to extend and optionally log events without requiring knowledge of actual TD command. The extend operation will occur even if the function cannot create an event log entry.

Status Code Returned

EFI_SUCCESS	Operation completed successfully.
EFI_INVALID_PARAMETER	One or more of the parameters are incorrect.
EFI_DEVICE_ERROR	The command was unsuccessful.
EFI_VOLUME_FULL	The extend operation occurred, but the event could not be written to one or more event logs.
EFI_UNSUPPORTED	The PE/COFF image type is not supported.

Related Definitions

```
//
// This bit is shall be set when an event shall be extended
// but not logged.
//
#define EFI_TD_FLAG_EXTEND_ONLY    0x0000000000000001

//
// This bit shall be set when the intent is to measure
// a PE/COFF image.
//
#define EFI_TD_FLAG_PE_COFF_IMAGE  0x0000000000000010

#pragma pack(1)

typedef struct {
  //
  // Size of the event header itself.
  //
  UINT32            HeaderSize;
  //
  // Header version. For this version of this specification,
  // the value shall be 1.
  //
  UINT16            HeaderVersion;
  //
  // Index of the MR that shall be extended.
  //
  UINT32            MrIndex;
  //
```

```
  // Type of the event that shall be extended
  // (and optionally logged).
  //
  UINT32            EventType;
} EFI_TD_EVENT_HEADER;

typedef struct {
  //
  // Total size of the event including the Size component,
  // the header and the Event data.
  //
  UINT32                Size;
  EFI_TD_EVENT_HEADER   Header;
  UINT8                 Event[1];
} EFI_TD_EVENT;

#pragma pack()
```

EFI_TD_PROTOCOL.MapPcrToMrIndex

Summary

This service provides callers information on TPM PCR to TDX measurement register (MR) mapping.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_TD_MAP_PCR_TO_MR_INDEX)(
  IN EFI_TD_PROTOCOL                    *This,
  IN UINT32                             PcrIndex,
  OUT UINT32                            *MrIndex
  );
```

Parameters

This

The protocol interface pointer.

PcrIndex

TPM PCR index.

MrIndex

TDX Measurement Register index.

Description

This function provides callers information on TPM PCR to TDX measurement register (MR) mapping.

In current version, we use below mapping:

TPM PCR Index	TDX Measurement Register Index	TDX-measurement register
0	0	MRTD
1, 7	1	RTMR[0]
2~6	2	RTMR[1]
8~15	3	RTMR[2]

Status Code Returned

EFI_SUCCESS	Operation completed successfully.
EFI_INVALID_PARAMETER	One or more of the parameters are incorrect.
EFI_DEVICE_ERROR	The command was unsuccessful.
EFI_VOLUME_FULL	The extend operation occurred, but the event could not be written to one or more event logs.
EFI_UNSUPPORTED	The PE/COFF image type is not supported.

EFI TD Final Events Table

All events generated after the invocation of **GetEventLog** SHALL be stored in an instance of an EFI_CONFIGURATION_TABLE named by the VendorGuid of **EFI_TD_FINAL_EVENTS_TABLE_GUID**. The associated table contents SHALL be referenced by the VendorTable of **EFI_TD_FINAL_EVENTS_TABLE**.

```
#define EFI_TD_FINAL_EVENTS_TABLE_GUID \
{0xdd4a4648, 0x2de7, 0x4665, {0x96, 0x4d, 0x21, 0xd9, 0xef, 0x5f, 0xb4
, 0x46}}


typedef struct {
  //
  // The version of this structure. It shall be set ot 1.
  //
  UINT64                  Version;
  //
  // Number of events recorded after invocation of GetEventLog API
  //
  UINT64                  NumberOfEvents;
  //
```

```
  // List of events of type TD_EVENT.
  //
  //TD_EVENT              Event[1];
} EFI_TD_FINAL_EVENTS_TABLE;


#pragma pack(1)

//
// Crypto Agile Log Entry Format.
// It is similar with TCG_PCR_EVENT2 except MrIndex.
//
typedef struct {
  UINT32               MrIndex;
  UINT32               EventType;
  TPML_DIGEST_VALUES   Digests;
  UINT32               EventSize;
  UINT8                Event[1];
} TD_EVENT;

#pragma pack()
```

4.3.3 TD-Event Log

TDVF may set up an ACPI table to pass the event-log information. The event log created by the TD owner contains the hashes to reconstruct the MRTD and RTMR registers.

Table 4-4: Intel TDX-Event-Log, ACPI Table

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'TDEL' Signature.
Length	4	4	Length, in bytes, of the entire Table
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	Standard ACPI header
OEM Table ID	8	16	Standard ACPI header
OEM Revision	4	24	Standard ACPI header
Creator ID	4	28	Standard ACPI header
Creator Revision	4	32	Standard ACPI header

Reserved	4	36	Reserved. Must be 0.
Log-Area-Minimum Length (LAML)	8	40	Identifies the minimum length (in bytes) of the system's pre-boot-TD-event-log area
Log-Area-Start Address (LASA)	8	48	Contains the 64-bit-physical address of the start of the system's pre-boot-TD-event-log area in QWORD format. Note: The log area ranges from address LASA to LASA+(LAML-1).

4.4 Storage-Volume-Key Data

In TD-execution environment, the storage volume will typically be an encrypted volume. In that case, by design, the TD-Guest Firmware will need to support quote generation and attestation to be able to fetch a set of storage-volume key(s) from a remote-key server during boot and pass the key to the guest kernel. Also by design, the key is stored in the memory, and the information of the key is passed from TD-Guest Firmware via an ACPI table (proposed below).

Table 4-5: Storage-Volume-Key-Location-ACPI Table

Field	Byte Length	Byte Offset	Description
Header			
Signature	4	0	'SVKL' Signature.
Length	4	4	Length, in bytes, of the entire Table
Revision	1	8	1
Checksum	1	9	Entire table must sum to zero.
OEMID	6	10	Standard ACPI header
OEM Table ID	8	16	Standard ACPI header
OEM Revision	4	24	Standard ACPI header
Creator ID	4	28	Standard ACPI header
Creator Revision	4	32	Standard ACPI header
Key Count (C)	4	36	The count of key structure
Key Structure	16 * C	40	The key structure

Table 4-6: Storage-Volume-Key Structure

Field	Byte Length	Byte Offset	Description
Key Type	2	0	The type of the key. 0: the main storage volume key 1~0xFFFF: reserved.
Key Format	2	2	The format of the key. 0: raw binary. 1~0xFFFF: reserved.
Key Size	4	4	The size of the key in bytes.
Key Address	8	8	The guest-physical address (GPA) of the key. The address must be in ACPI-Reserved Memory.

5 TD-VMM-Communication Scenarios

5.1 Requesting IPIs

Various. TD-VMM-communication scenarios require the TD to request the host generate IPIs to TD vCPUs – for example, synchronizing, guest-TD-kernel-managed-IA-page-table updates. This operation is supported via the TDCALL [TDG.VP.VMCALL <Instruction.WRMSR>] to the x2APIC ICR MSR.

To perform a cross-VCPU IPI, the guest-TD ILP is designed to request an operation from the host VMM using this TDCALL (TDG.VP.VMCALL <Instruction.WRMSR>). The VMM can then inject an interrupt into the guest TD's RLPs using the posted-interrupt mechanism. This is an untrusted operation; thus, the TD OS must track its completion and not rely on the host VMM to faithfully deliver IPIs to all the TD vCPUs.

5.2 TD-memory conversion and memory ballooning

Recall that, by design, guest-physical memory used by a TD is encrypted with a TD-private key or with a VMM-managed key based on the GPA-shared bit (GPA [47 or 51] based on GPAW). A TD OS may operate on a fixed, private-GPA space configured by the host VMM. Typically, the OS manages a physical-page-frame database for state of (guest) physical-memory allocations. Instead of expanding these PFN databases for large swaths of shared-GPA space, the TD OS can manage an attribute for the state of physical memory to indicate whether it is encrypted with the TD-private key or a VMM key. Such a TD-guest OS can use TDG.VP.VMCALL(MapGPA) so that, within this fixed-GPA map, the TD OS can request the host VMM map Shared-IO memory aliased as shared memory in that GPA space - so in this case, the OS can select a page of the private-GPA space and make a TDG.VP.VMCALL(MapGPA(GPA) with GPA.S=1) to map that GPA using the S=1 alias. The VMM can then TDH.MEM.RANGE.BLOCK, TDH.MEM.TRACK, and TDH.MEM.SEPT.REMOVE the affected GPA from the S-EPT mapping; and the VMM can then re-claim the page using direct-memory stores and map the alias-shared GPA for the TD OS in the shared EPT (managed by the VMM).

At a later point, the TD OS may desire to use the GPA as a private page via the same TDG.VP.VMCALL(MapGPA) with the GPA specified as a private GPA (GPA.S=0) – the intent is for this to allow the host VMM to unlink the page from the Shared EPT and then perform a TDH.MEM.PAGE.AUG to set up a pending-EPT mapping for the private GPA. The successful completion of the TDG.VP.VMCALL flow can be used by the TD guest to TDG.MEM.PAGE.ACCEPT to re-initialize the page using the TD-private key and mark the S-EPT mapping as active.

5.3 Paravirtualized IO

The TD guest can use paravirtualized-IO interfaces (for example, using virtio API in KVM) exposed by the host VMM to use physical and virtual devices on the host platform that are managed by the VMM. For this scenario, Virtualized IO is typically enumerated over emulated PCIe (port I/O or MMIO). The TD drivers can help ensure that the data passed via memory referenced in emulated-MMIO accesses are placed in the TD's shared-GPA-memory space. Paravirtualized drivers could pre-allocate a primary-

shared buffer during initialization. Subsequently, drivers can allocate a portion of the shared-GPA-space buffer for each individual transfer and reclaim the buffer after a specific transfer is completed. In this scenario, the primary-buffer can expand and shrink as needed. Shared buffers can be deallocated during driver-stack tear-down. This scenario is optimal, as allocating shared buffer can involve at least one TDG.VP.VMCALL (for mapping shared page) and TDCALL[TDG.MEM.PAGE.ACCEPT] for mapping back as a private-TD page, as described in Section 4.3.

The guest TD may employ VMM functions for IO to participate in the emulation of MMIO accesses from legacy-device drivers. To support this scenario, if the TD OS opts-in, the host VMM can host the emulated-device-MMIO space in shared-GPA space of the TD OS. Legacy-device-driver accesses to the emulated region can cause EPT violations that can be mutated to the TD-#VE handler that can then support emulation of the MMIO. The enlightened-TD-OS-#VE handler can emulate the access causing the #VE by decoding the instruction (within the TD) and invoking the Instruction.IO functions hosted by the VMM using TDCALL [TDG.VP.VMCALL <Instruction.MMIO>]. From that point on, like the previous paravirtualized IO model, the TD software must ensure that the data buffers passed via memory referenced in parameters that are passed in function TDG.VP.VMCALL are placed in the TD's shared -GPA space.

5.4 TD attestation

Goals of TD Attestation are to enable the TD OS to request a TDREPORT that contains version information about the Intel TDX module, measurement of the TD, along with a TD-specified nonce. By design, the TDREPORT is locally MAC'd and used to generate a quote for the TD via a quoting enclave (QE). The remote verifier can verify the quote to help verify the trustworthiness of the TD.

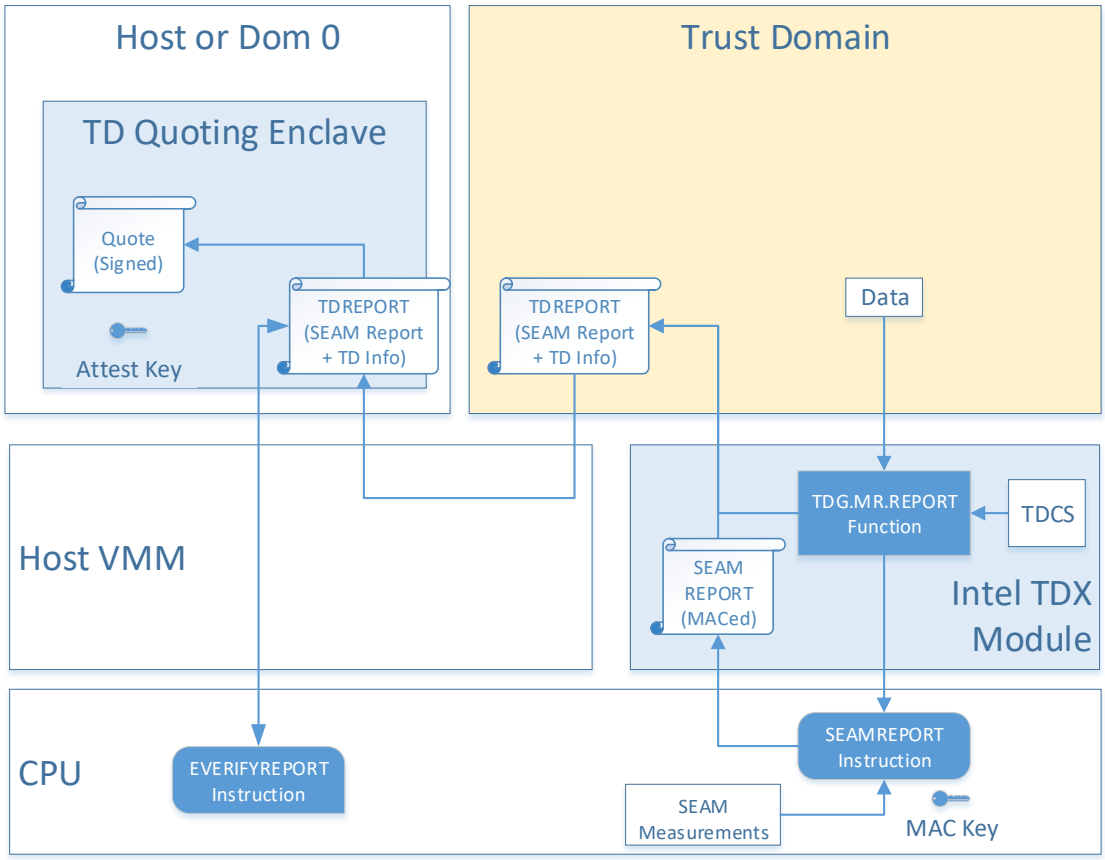

Figure 5-1: TD-Attestation flow

1. Guest-TD software invokes the TDCALL(TDG.MR.REPORT)-API function.
2. The Intel TDX module uses the SEAMOPS[SEAMREPORT] instruction to create a MAC'd TDREPORT_STRUCT with the Intel TDX-module measurements from CPU and TD measurements from the TDCS.
3. Guest-TD software uses the TDCALL(TDG.VP.VMCALL) interface to request the TDREPORT_STRUCT be converted into a Quote.
4. The TD-Quoting enclave uses ENCL[EVERIFYREPORT2] to verify the TDREPORT_STRUCT. This allows the Quoting Enclave to help verify the report without requiring direct access to the CPU's HMAC key. Once the integrity of the TDREPORT_STRUCT has been verified, the TD-Quoting Enclave signs the TDREPORT_STRUCT body with an ECDSA-384-signing key.
5. The Quote can then be used by TD software to perform a remote-attestation protocol with a verifying-remote party.

5.5 Service TD Binding

Service TD is a Trust Domain (TD) VM used to provide a dedicated service/utility. The service TD extends the TCB of the tenant TD which the service TD provides the service to. Migration TD (MigTD) is an example Service TD.

One or more **service TDs** may be bound to a **target TD**. Service TD binding relationship has the following characteristics:

- A service TD has a **type** (SERVTD_TYPE).
- A service TD may **read and/or write certain target TD metadata**. Access permission to target TD metadata fields depends on SERVTD_TYPE.
- **Unsolicited service TD binding** is done without target TD approval. The target TD needs not be aware of the binding.
- The target TD's TDREPORT indicates binding to service TDs.
- The service TD protocol consists of:
  - Binding
  - Metadata access
- Service TD to target TD binding relationship is many-to-many
  - Multiple service TDs of different types may be bound to a single target TD.
  - Multiple target TDs may be bound to a single service TD.
- A service TD may itself be a target TD to other service TDs.

Typical Unsolicited Service TD Binding and Metadata Access Use Case

1. **Optional Pre-Binding:** During target TD build, before calling SEAMCALL[TDH.MR.FINALIZE], the host VMM calls SEAMCALL[TDH.SERVTD.PREBIND] to write the binding fields (SERVTD_HASH etc.) in the target TD's service TD table.
2. **Binding:** Sometime later, the host VMM calls SEAMCALL[TDH.SERVTD.BIND] to bind the service TD. It gets back a binding handle. The VMM communicates the binding handle, target TD_UUID and other binding parameters to the service TD.
3. **Metadata Access:** The service TD uses TDCALL[TDG.SERVTD.RD and TDG.SERVTD.WR] to access target TD metadata.
4. **Rebinding:** May be required due to, e.g., both target TD and service TD have been migrated or a new service TD instance replaces the original one. The host VMM calls SEAMCALL[TDH.SERVTD.BIND] to rebind the service TD. It gets back a binding handle. The VMM communicates the binding handle, target TD_UUID and other binding parameters to the service TD.

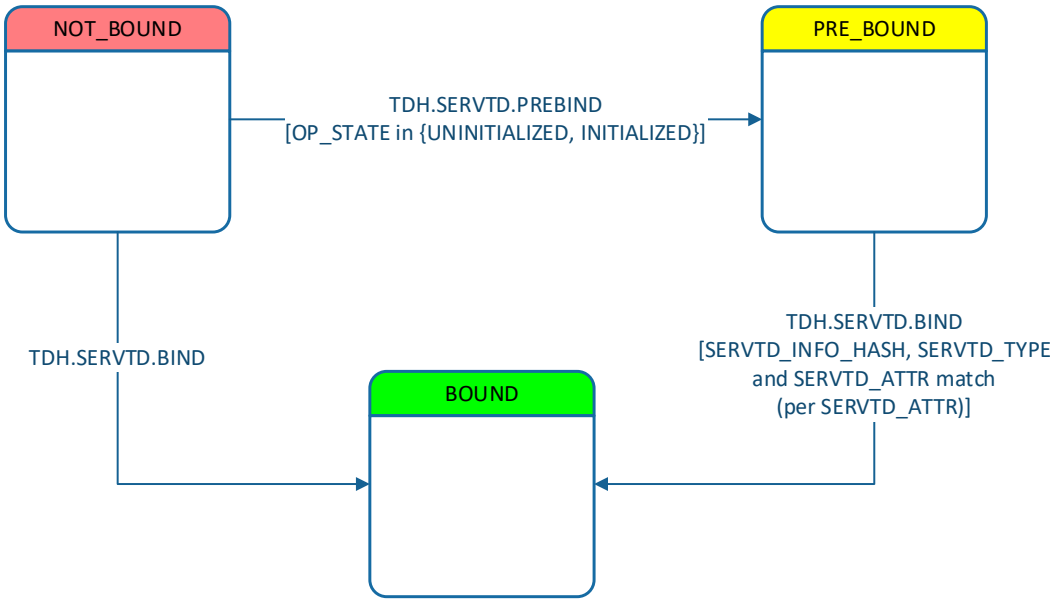

Figure 5-2: Service TD Binding State Machine

For more detail on service TD, please refer to Intel TDX Module Specification.

5.6 TD Live Migration

Analogous to legacy VM migration, a cloud-service provider (CSP) may want to relocate/migrate an executing Trust Domain from a **source TDX platform** to a **destination TDX platform** in the cloud environment. A cloud provider may use TD migration to meet customer Service Level Agreement (SLA), while balancing cloud platform upgradability, patching and other serviceability requirements. Since a TD runs in a CPU mode which protects the confidentiality of its memory contents and its CPU state from any other platform software, including the hosting Virtual Machine Monitor (VMM), this primary security objective should be maintained while allowing the TD resource manager, i.e., the host VMM to migrate TDs across compatible platforms. The TD typically may be assigned a different HKID (and will be always assigned a different ephemeral key) on the destination platform chosen to migrate the TD.

The TD being migrated is called the **source TD**, and the TD created as a result of the migration is called the **destination TD**. An extensible **TD Migration Policy** is associated with a TD that is used to maintain the TD's security posture. The TD Migration policy is enforced in a scalable and extensible manner using a specific type of **Service TD** called the **Migration TD (a.k.a. MigTD)** – which is used to provide services for migrating TDs.

The TD Live Migration process (and the Migration TD) does not depend on any interaction with the TD guest software operating inside the TD being migrated.

Figure 5 shows the lifecycle of a TD Live Migration process and the corresponding Intel TDX Module APIs involved.

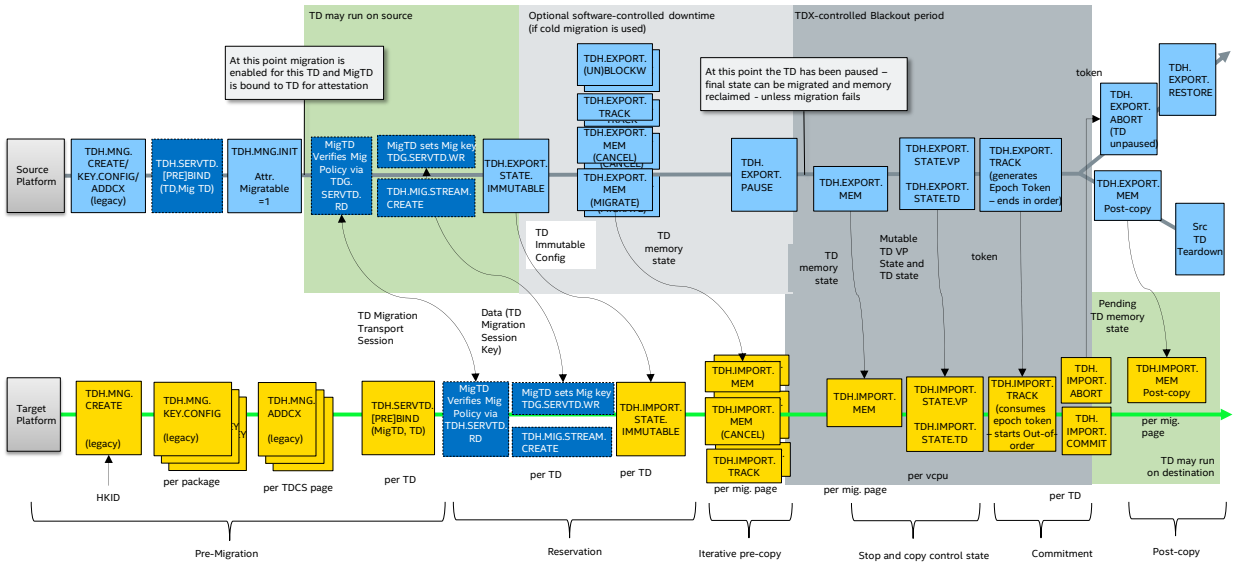

Figure 5-3: TD Migration Lifecycle Overview

5.6.1 Pre-Migration

5.6.1.1 Intel TDX Module Enumeration

The host VMM uses SEAMCALL[TDH.SYS.RD] or SEAMCALL[TDH.SYS.RDALL] to enumerate Intel TDX Module functionality and learns from the TDX_FEATURES that the Intel TDX Module supports TD Migration.

5.6.2 Reservation and Session Setup

5.6.2.1 Source Guest TD Build and Execution

The host VMM uses SEAMCALL[TDH.MNG.INIT] function to initialize a TD, with **ATTRIBUTES.MIGRATABLE** bit set to 1.

Before a migration session can begin, the VMM on the source platform must use SEAMCALL[TDH.SERVTD.BIND] to bind a Migration TD to the source TD.

5.6.2.2 Destination Guest TD Initial Build

The host VMM creates a new guest TD by using the SEAMCALL[TDH.MNG.CREATE] function. This destination TD is setup as a “template” to receive the state of the Source Guest TD.

The host VMM programs the HKID and HW-generated encryption key assigned to the TD into the MKTME encryption engines using the SEAMCALL[TDH.MNG.KEY.CONFIG] function on each package.

The host VMM can then continue to build the TDCS by adding TDCS pages using the SEAMCALL[TDH.MNG.ADDCX] interface function.

Once the destination TDCS is built and before TD import can begin, the VMM on the destination platform must use SEAMCALL[TDH.SERVTD.BIND] to bind a Migration TD to the destination TD.

5.6.2.3 Migration Session Key Negotiation

The host VMMs in source platform and destination platform notify the MigTD on the source and destination platform on the migration request.

The MigTDs executing on the source and destination platforms use a TD-quote-based mutual authentication protocol to create a VMM-transport-agnostic secure session between them, such as remote-attestation TLS (RA-TLS). Using this secure session, the migration policy can be evaluated by the MigTDs. After policy check, the Migration TD transfer the Migration Session Key (MSK) to the peer. The MSK is an ephemeral AES-256-GCM key used for confidentiality and integrity of the TD private state exported from the source platform and imported on the destination platform.

The Service TD binding mechanism supported by the TDX module allows the Migration TD to access target TD metadata – specifically the Migration Session Key. The MigTD can read/write the TD metadata using TDCALL[TDG.SERVTD.RD/WR*] guest-side interface functions. The Migration TDs on both the source and destination platforms must use this interface to read/write the Migration Session Key (as meta-data) to the target TD's control structures.

After this point, the host VMM can invoke TDX Module functions such as SEAMCALL[TDH.EXPORT.*] to export state at the source platform and SEAMCALL[TDH.IMPORT.*] to import TD state at the destination platform.

5.6.2.4 TD Global Immutable Metadata (Non-Memory State) Migration

Immutable metadata is the set of TD state variables that are set by SEAMCALL[TDH.MNG.INIT], may be modified during TD build but are never modified after the TD's measurement is finalized using SEAMCALL[TDH.MR.FINALIZE]. Some of these state variables control how the TD and its memory is migrated. Therefore, the immutable TD control state is migrated before any of the TD memory state is migrated.

The host VMMs use SEAMCALL[TDH.EXPORT.STATE.IMMUTABLE] to export TD immutable state at the source platform and use SEAMCALL[TDH.IMPORT.STATE.IMMUTABLE] to import TD immutable state at the destination platform.

5.6.3 Iterative Pre-Copy of Memory State

5.6.3.1 Migration Considerations for TD Private Memory

Intel TDX protects guest TD state in private memory from a malicious VMM, using MKTME (memory encryption and integrity protection) and the Intel TDX Module. The Intel TDX Module performs ephemeral key id management to enforce the TDX security objectives. Memory encryption is performed by encryption engines that reside at each memory controller, with no software access (including the TDX module) to the ephemeral keys. The memory encryption engine holds a table of encryption keys, in the Key Encryption Table (KET). The encryption key selected for memory transactions is based on a Host Key Identifier (HKID) provided with the memory access transaction.

The Intel TDX Module API functions enable the host VMM to manage HKID assignment to guest TDs, configure the memory encryption engines etc., while assuring proper operation to maintain TDX's security objectives. The host VMM also does not have access to the TD encryption keys.

TD Migration does not migrate the HKIDs – a free HKID is assigned to the TD created on the destination platform to receive migratable assets of the TD from the source platform. All TD private memory is protected during transport from the source platform to the destination platform using an intermediate encryption performed using the MSK negotiated via the Migration TDs on the source and destination platform. On the destination platform the memory is encrypted via the destination ephemeral key as it is imported into the destination platform memory assigned to the destination TD. The import operation on the destination TDX module verifies and decrypts the TD private data using the MSK, and uses the MKTME engine to encrypt (and integrity protect) while writing it to memory using the destination TD HKID.

Shared memory assigned to the TD is migrated using legacy mechanisms used by the host VMM.

5.6.3.2 Migration Considerations for EPT Structures

Guest Physical Address (GPA) space is divided into private and shared sub-spaces, determined by the SHARED bit of GPA. The CPU translates shared GPAs using the Shared EPT, which resides in host VMM memory, and is directly managed by the host VMM, same as with legacy VMX. The CPU translates private GPAs using a separate Secure EPT. Secure EPT pages are encrypted, and integrity protected with the TD's ephemeral private key.

As there is no guarantee of allocating the same physical memory addresses to the TD being migrated on the destination platform, **the memory used for Secure EPT structures is not migrated across platforms**. Hence, the VMM must invoke the TDX module's SEAMCALL[TDH.MEM.SEPT.*] interface functions on the destination platform to re-create the private GPA mappings on the destination platform (per the assigned HPAs). The Intel TDX module uses the cryptographically protected exported meta-data (generated via SEAMCALL[TDH.EXPORT.MEM]) to verify and enforce (via the SEAMCALL[TDH.IMPORT.MEM]) that the Secure EPT security properties from the source platform are rec-created correctly as TD private memory contents are migrated, thus preventing remap attacks during migration.

Even though Secure EPT structures are not migrated, the source SEPT structures track the state of the mappings when a page is exported and then modified by the TD OS in the pre-copy stage. The TD OS may be allowed to modify such a page and the TDX module enforces that the modified and previously exported page is re-exported by the source host VMM and re-imported by the destination host VMM.

5.6.3.3 Post Copy: Destination Guest TD Execution during Memory Migration

In a typical live migration scenario, the TD is expected to resume executing on the destination platform shortly after it is paused on the source platform. The destination TD can only begin executing after the pre-copy stage completes and the destination TD control state has been imported – memory transfer may continue after that in a post-copy stage. Pre-copy stage imports the working set of memory pages, the host VMM must have paused the source TD, exported the final mutable control state and imported the final mutable control state to the destination TD virtual processors and control state. The Intel TDX module enforces the security objectives of this Commitment protocol, with the remaining memory state transferred in the post-copy stage which also happens via TDX Module interfaces – SEAMCALL[TDH.EXPORT.MEM] and SEAMCALL[TDH.IMPORT.MEM].

5.6.4 Source TD Stop and Final Non-Memory State Migration

Following pre-copy of TD private memory, the host VMM must pause the source TD for a brief period (also called the blackout period) so that the VMM may export the final control state (for all VCPUs and for the TD overall). The VMM initiates this via SEAMCALL[TDH.EXPORT.PAUSE], which checks security pre-conditions and prevents TD VCPUs from executing any more. It then allows export of final (mutable) TD non-memory state.

5.6.4.1 Final Memory State Migration

The host VMMs use SEAMCALL[TDH.EXPORT.MEM] and SEAMCALL[TDH.IMPORT.MEM] to migrate memory contents during this source (and destination) TD paused state. The TDX Module enforces that all exported state for the source TD must be imported before the destination TD may run using the commitment protocol described below.

5.6.4.2 TD-Scope and VCPU-Scope Mutable Non-Memory State migration

TD mutable non-memory state is a set of source TD state variables that might have changed since it was finalized via SEAMCALL[TDH.MR.FINALIZE]. Immutable non-memory state exists for the TD scope (as part of the TDR and TDCS control structures) and the VCPU scope (as part of the TDVPS control structure).

Mutable TD state is exported by SEAMCALL[TDH.EXPORT.STATE.TD] (per TD) and SEAMCALL[TDH.EXPORT.STATE.VP] (per VCPU) and imported by SEAMCALL[TDH.IMPORT.STATE.TD] and SEAMCALL[TDH.IMPORT.STATE.VP] respectively.

5.6.5 Commitment

The commitment protocol is enforced by the Intel TDX Module to help ensure that a host VMM cannot violate the security objectives of TD Live migration.

This protocol is enforced via the following TDX Module interface functions:

- On the source platform, SEAMCALL[TDH.EXPORT.PAUSE] starts the blackout phase of TD live migration and SEAMCALL[TDH.EXPORT.TRACK] ends the blackout phase of live migration (and marks the end of the transfer of TD memory pre-copy, mutable TD VP and mutable TD global control state). SEAMCALL[TDH.EXPORT.TRACK] generates a MSK-based cryptographically-authenticated start token to allow the destination TD to become runnable. On the destination platform, SEAMCALL[TDH.IMPORT.TRACK] – which consumes the cryptographic start token, allows the destination TD to be un-paused.
- In error scenarios, the migration process may be aborted proactively by the host on the source platform via SEAMCALL[TDH.EXPORT.ABORT] before a start token was generated; if a start token was already generated (i.e. pre-copy completed), the destination platform can generate an abort token using SEAMCALL[TDH.IMPORT.ABORT] which generates an abort token which may be consumed by SEAMCALL[TDH.EXPORT.ABORT] by the source TD platform TDX Module to abort the migration process and again allows the source TD to become runnable again.

5.6.6 Post-Copy of Memory State

In some live migration scenarios, the host VMM may stage some memory state transfer to occur lazily after the destination TD has started execution. In this case, the host VMM will be required to fetch the

required pages as accesses occur by the destination TD – this order of access is indeterminate and will likely differ from the order in which the host VMM has queued memory state to be transferred.

In order to support that on-demand model, the order of memory migration during this **post-copy stage** is not enforced by TDX. The host VMM may implement multiple migration queues with multiple priorities for memory state transfer. For example, the host VMM on the source platform may keep a copy of each encrypted migrated page until it receives a confirmation from the destination that the page has been successfully imported. If needed, that copy can be re-sent on a high priority queue. Another option is, instead of holding a copy of exported pages, to call SEAMCALL[TDH.EXPORT.MEM] again on demand.

Also, to simplify host VMM software for this model, the TDX module interface functions used for memory import in this post-copy stage return additional informational error codes to indicate that a stale import was attempted by the host-VMM to account for the case where the low-latency import operation for a GPA superseded the import from the higher latency import queue.

For more detail on TD Live Migration, please refer to Intel TDX Module TD Migration Specification and Migration TD design guide