Intel® TDX Virtual Firmware Design Guide

December 2022

Document Number: 344991-003US

Document Number: 344991-003US

Contents

Figures

Tables

1 Introduction

1.1 Background

Intel Total Memory Encryption (TME) engine encrypts the platform's entire memory with a single key and provides the ability to specify use of a specific key for a page of memory. The Multi-Key mode of TME (MK-TME) extends TME to support multiple encryption keys. In a virtualization scenario, a Virtual Machine Manager (VMM) or hypervisor will manage keys to transparently support legacy operating systems without any changes (thus, MKTME can also be viewed as TME virtualization in these scenarios). An operating system (OS) may take advantage of MKTME capabilities in a native or virtualized environment. When properly enabled, MKTME is available to each guest OS in a virtualized environment, so both native and guest OS can take advantage of MKTME. In these usages, the VMM is in the Trust Computing Base (TCB).

Intel Trust Domain Extensions (Intel TDX) refers to 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 a Trust Domain (TD). A TD runs in a CPU mode that protects the confidentiality of its memory contents and its CPU state from any other software, including the hosting Virtual Machine Monitor (VMM), unless explicitly shared by the TD itself.

The TDX solution is built using a combination of Intel® Virtual Machine Extensions (VMX) and Multi-Key Total Memory Encryption (MK-TME), as extended by the Intel® Trust Domain Instruction Set Architecture Extensions (TDX ISA). An attested software module called The Intel TDX module implements the TDX architecture.

The platform is managed by a TDX-aware host VMM. A host VMM can launch and manage both guest TDs and legacy guest VMs. The host VMM maintains all legacy functionality from the legacy VMs perspective; it's restricted only with regard to the TDs it manages.

In Summary, Intel TDX:

1) Removes the cloud host software and devices from the Trust Computing Base (TCB) of cloud (TD) tenants.

2) Provides memory encryption and integrity multi-tenancy for hardware attack protection.

3) Supports TD measurement for attestation to a relying party.

1.2 Overview

Intel® Trust Domain Extensions (Intel® TDX) provides the capabilities required to enable an TDX-aware VMM to manage the lifecycle of a TD.

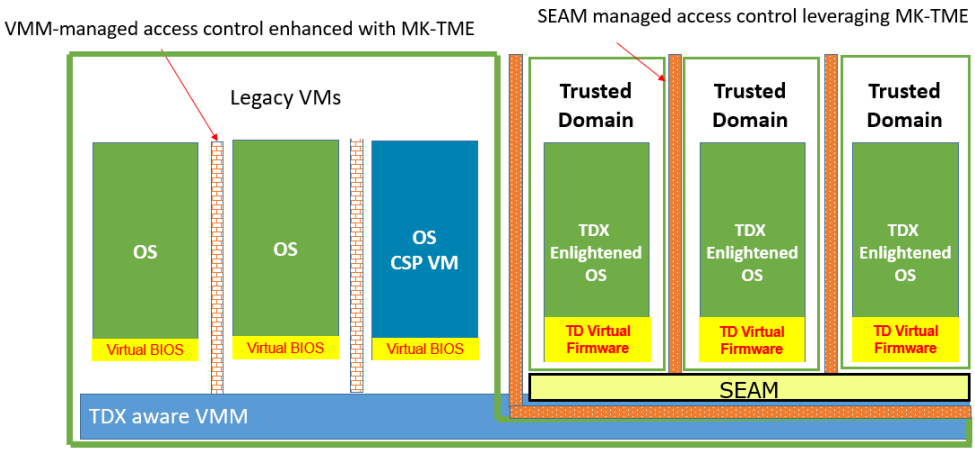

Figure 1-1: Intel TDX Architecture

Fundamental differences between Virtual Machine Extension (VMX) and Intel TDX are outlined in the following table.

Table 1-1: Differences between VMX and TDX

	Access to Guest State	Transitions for State Save & Restore	Controls
VMX	VMM has full access	VMM SW augments VMX transitions (e.g. general-purpose registers)	VMM has a rich set of controls to interact with the VM in order to de-privilege the VM
TDX	VMM has no direct access; TD may volunteer info	State transitions completed are managed by Intel TDX Module in Secure-Arbitration Mode (SEAM)	VMM has a limited set of controls for resource management

Trust Domain Virtual Firmware (TDVF) is required to provide TD services to the TD Guest OS. This document describes the TDVF architecture.

TDVF reference code is integrated to TianoCore open source repository: https://github.com/tianocore/edk2/tree/master/OvmfPkg. A minimal TDVF (td-shim) reference code is at confidential container repository: https://github.com/confidential-containers/td-shim.

1.3 Terminology

Table 1-2: Terminology

Term	Description
ACPI	Advanced Configuration and Power Interface

AP	Application Processor
APIC	Advanced Programmable Interrupt Controller
BFV	Boot Firmware Volume
BSP	Bootstrap Processor
CFV	Configuration Firmware Volume
CSM	Compatibility Support Module
DXE	Driver Execution Environment
EPT	Extended Page Table
GPA	Guest Physical Address
GPAW	Guest Physical Address Width
HOB	Hand Off Block
HPA	Host Physical Address
MADT	Multiple APIC Description Table
MKTME	Multi-Key Total Memory Encryption
MPWK	Multiple Processor Wakeup
MRCONFIGID	Measurement Register of configuration data
MRTD	TD Measurement Register
OVMF	Open Virtual Machine Firmware
PEI	Pre-EFI Initialization
RA-TLS	Remote Attestation-TLS
RTMR	Runtime Extendable Measurement Register
SEAM	Secure-Arbitration Mode
SEC	Security Phase
SMM	System Management Mode
SMX	Safer Mode Extensions
TCB	Trust Computing Base
TCG	Trusted Computing Group
TD	Trust Domain
TDVF	Trust Domain Virtual Firmware
TDX	(Intel) Trust Domain Extensions
TLS	Transport Layer Security
TME	Total Memory Encryption
UEFI	Unified Extensible Firmware Interface
VCPU	Virtual CPU
VE	Virtualization Exception
VM	Virtual Machine

VMCS	Virtual Machine Control State
VMM	Virtual Machine Monitor
VMX	Virtual Machine Extension

§

2 Architectural Overview

2.1 TDVF Features

TDVF has the following features:

1) Use Unified Extensible Firmware Interface (UEFI) Secure Boot as base with extensions for TD launch.

2) Use Trusted Computing Group (TCG) Trusted Boot to perform a measured and verified launch of a guest OS loader or kernel.

3) Simplify firmware by removing features found in traditional UEFI implementations:

    a) SEC, PEI, SMM (DXE Only)

    b) CSM (UEFI Class 3 OS only)

    c) Setup UI

    d) Recovery

    e) Capsule-based Firmware Update

    f) ACPI S3 (not supported by TDX guests)

2.2 TD Hardware

A TD is based on the following hardware:

1) CPU: x2APIC

2) CPU: xFLUSH, STI, CLI, LIDT, LGDT instruction are allowed.

3) VMM-specific Virtual Device (block device, console, network). This is highly dependent on the hypervisor configuration (e.g. VirtIo device for KVM/XEN, Vmbus device for Microsoft Hyper-V).

4) Hot Plug – CPU and memory hotplug are not supported now. Device hotplug is out of scope of this document.

5) TD may or may not support below feature. If it is supported, the device must be access via TDCALL [TDG.VP.VMCALL] interface instead of direct hardware access. For example, the IO access needs to be replaced by TDCALL [TDG.VP.VMCALL] <INSTRUCTION.IO>.

    a) Persistent NV storage.

    b) The emulated physical device. (Graphic, Keyboard, Storage, etc.)

c) I/O Subsystems (PCI, USB, ISA, DMA, IOMMU, PIC, PIT, etc.). For example, the PCI might only be used to emulate the VirtIo device.

d) MMIO (APIC, HPET)

e) vTPM

2.3 Boot Flow

A TD launch takes below steps:

1) VMM sets up TDVF, calls Intel TDX module to create the initial measurement, then calls Intel TDX module to launch TDVF.

2) TDVF boots and enables UEFI Secure Boot.

3) TDVF prepares TD event log and launches the OS loader.

2.3.1 VMM Setup Phase

The TDVF image includes firmware code, which is measured into TD Measurement Register (MRTD). The TDVF image may also include static configuration data, including UEFI Secure Boot certificates (PK/KEK/db/dbx). The VMM provides dynamic configuration data, including the hand-off block (HOB) list as a parameter for the entrypoint.

The VMM calls Intel TDX module to initialize TD memory. This includes firmware code and UEFI Secure Boot configuration read-only data captured by the tenant:

- Intel TDX module associates memory via Extended Page Table (EPT) with the TD guest.
- Intel TDX module associates logical processors with TD guest via TD-Virtual Machine Control State (VMCS).
- Intel TDX module performs TDENTER instruction on all processors, including Bootstrap Processor (BSP) and Application Processors (APs).

2.3.2 TDVF Launch Phase

TDVF is launched on all processors:

- All processors start in 32-bit protected mode with flat descriptors (paging disabled).
- The CPU with VCPU_INDEX 0 is elected as BSP, the other CPUs are APs.
- Startup code switches to 4-level paging enabled (0-4GB). Option for startup code to switch to 5-level paging enabled (0-4GB).
- BSP performs Virtual Firmware initialization and determines how many APs to wake via TDCALL [TDG.VP.INFO].

- APs perform TDCALL [TDG.VP.INFO] and wait for virtual-wake triggered in-memory by BSP.

2.3.3 TDVF OS Boot Phase

The TDVF prepared information and boots to OS loader finally.

- Memory map is prepared, and private memory is used.
- ACPI Tables report platform information.
- UEFI Secure Boot is enabled.
- TD event log is prepared.
- APs are in wait for wake-up.

2.4 TDVF Requirements

TDVF should meet the following requirements:

1) TDVF is launched by a hypervisor and Intel TDX module (see 'TD Launch' for details).
   a) The entry point of TDVF is 32-bit protected mode, launched by Intel TDX module.
   b) TDVF enables long mode and continues to run in long mode.
   c) TDVF parses system information passed from the hypervisor.
   d) TDVF halts the AP till AP wakeup.
2) TDVF launches guest TD.
   a) TDVF starts the guest TD OS loader.
   b) TDVF provides memory map to guest TD (see 'TD Memory Management' for details).
   c) TDVF provides ACPI table to guest TD (see 'TDVF ACPI Support' for details).
   d) TDVF supports multi-processors and allows guest TD to wake up APs. (see 'TD Launch' for details).
3) Security
   a) TDVF enables UEFI Secure Boot (see 'TDVF UEFI Secure Boot Support' for details)
   b) TDVF creates the TD event log and pass it to guest TD (see 'TD Measurement' for details).
   c) TDVF sets up private memory (see 'TD Memory Management' for details).

2.5 Reproducibility

If a virtual firmware (TDVF or Td-shim) is used in a service TD, the binary should be reproducible, meaning you can recreate the same release binary with the same source code version and same compiler version. This condition should be true under any circumstances, anytime, on any machine, and any source path location.

2.5.1 Challenge

There are reproducibility challenges today. For example, some C code may use __FILE__, __LINE__, __DATE__, __TIME__ as the build-in information. The __FILE__ may change based on the file location. __DATA__ and __TIME__ may change based on the build time.

The compiler may generate a debug entry, including a source file path. The Rust compiler may add additional tool file paths, such as CARGO_HOME and RUSTUP_HOME.

For a PE/COFF image, the compiler may generate a time stamp for when the binary is created. In addition, the compiler may include the unique GUID to provide detailed debugging information.

Refer to the Microsoft PE/COFF specification for more details.

2.5.2 Solution

To resolve the reproducibility challenge, we have the following recommendation:

2.5.2.1 Code

The C code should use __FILE__, __LINE__, __DATE__, __TIME__ only for debug purposes.

If present, the info should be included in the DEBUG macro and removed in release build.

2.5.2.2 Compiler Option

Don’t generate any debug info in the release build. Strip all symbols.

2.5.2.3 Additional tools

The additional tools should be created to zero the debug data fields in the PE image, and zero the time stamp and unique GUID.

For example, the EDKII project (using C) includes a `GenFW` tool that has -z, --zero option to zero such information. The Td-shim project (using Rust) includes a `td-shim-strip-info` tool that can zero such information for a PE image and zero the Rust tool path.

2.6 Security Considerations

This section introduces security practices for the TD virtual firmware. The Linux kernel version is at https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html.

2.6.1 General Security Practice

2.6.1.1 Minimize the Attack Surface

The virtual firmware should include a driver only when it is necessary. For example, the TCP/IP network is a big attack surface, and it should be excluded in the TDVF.

2.6.1.2 Input Validation

Any input data from VMM should be treated as untrusted. The data must be verified before use. If the data is in shared memory, TDVF should copy the data to private memory, then validate and use.

2.6.1.3 Enable Defensive Technologies

TDVF should consider enabling the following technologies:

1) Data Execution Protection (DEP)

The virtual firmware may set up DEP to mark data regions (including stack and heap) to be non-executable, and mark code regions to be read-only to prevent code injection attacks.

2) Address Space Layout Randomization (ASLR)

The virtual firmware may use ASLR for the heap/stack allocation. The effectiveness of ASLR is based upon random bit entropy. If the TDVF only has limited memory resources, then the entropy bit cannot be too large. But shifting even a few bits offers some benefit.

3) Control Flow Enforcement Technology (CET)

The virtual firmware may set up CET Shadow Stack (SS) protection to prevent Return Oriented Programming (ROP) attacks.

The virtual firmware may set up CET Indirect Branch Tracking (IBT) to prevent Jump Oriented Programming (JOP) attacks. This mainly depends upon the compiler's capability.

2.6.2 Side Channel Security Practice

The TDVF should consider side channel attacks that may steal secrets, including but not limited to the disk encryption key, TLS session key, SPDM session key, ephemeral private key, etc. The host firmware version is at https://www.intel.com/content/www/us/en/developer/articles/technical/software-

security-guidance/technical-documentation/host-firmware-speculative-side-channel-mitigation.html.

2.6.2.1 Bound Check Bypass (Spectre Variant 1)

The TDVF should use LFENCE before parsing any VMM input data if the data includes length or offset field, such as CFV and TD_HOB.

Care must be taken that the data from the device might also be from VMM, including MMIO, IO, or PCI Configuration. The device driver should also follow the same rule to consume the length or offset fields.

2.6.2.2 Branch Target Injection (Spectre Variant 2)

N/A. Branch predictions cached by the CPU before entering a guest TD should not impact the behavior of that TD. The Intel TDX module helps ensure that by applying CPU mechanisms to isolate the branch predictions of each guest TD from branch predications done elsewhere.

2.6.2.3 Rogue Data Cache Load (Meltdown Variant 3)

N/A. The TDVF is a bare metal execution environment. All code runs in supervisor mode.

2.6.2.4 Rogue System Register Load (Meltdown Variant 3a)

N/A.

2.6.2.5 Speculative Store Bypass (Spectre Variant 4)

N/A.

2.6.3 Confidential Computing Virtual Firmware Security Practice

2.6.3.1 Measured Boot

TDVF should follow the measured boot principle to construct the chain of trust.

Any configuration data from VMM should be measured to a measurement register (MR) before being used, including but not limited to the data in CFV, TD_HOB (ACPI), or VMM specific configuration, such as QEMU FW_CONFIG_IO.

The dynamic data from VMM, such as device state or device base address register, should not be measured. These data should be treated as untrusted input and should be verified before use.

Please refer to chapter 8 TD Measurement for more detail.

2.6.3.2 Random Number

If a random number is required, TDVF should only use RDSEED/RDRAND CPU instruction.

Other software random number, such as virtio-rng is not trusted.

2.6.3.3 Timer

TDX architecture does not provide a trusted timer. The legacy source, such as Runtime-Clock (RTC) via port 0x70/0x71 is not trusted.

If a trusted timer is required, the TDVF should connect to an authenticated time server.

2.6.3.4 CPU Information

The TDVF may require accessing CPU information such as CPUID or MSR. Most of those resources are emulated by VMM, and they cannot be trusted.

Only a small portion of CPUID and MSR are provided to the TDX-module, and they can be trusted. Please refer to the TDX Module architecture specification, chapter 18 ABI reference: CPU Virtualization Tables.

2.6.3.5 Device Information

The TDVF may require accessing resources such as MMIO, IO, PCI Configuration Space, etc. Those resources are emulated by VMM, and they cannot be trusted. TDVF should verify the data before use.

2.6.4 TDX Specific Security Practice

2.6.4.1 Private Memory Acceptance

The TDVF needs to use TDCALL[TDG.MEM.PAGE.ACCEPT] to accept PENDING pages added by VMM via SEAMCALL[TDH.MEM.PAGE.AUG]

The TDVF should track the accepted pages and not accept the previously accepted memory. Otherwise, the VMM could zero-out a page by removing it and add a new one at the same address.

2.6.4.2 MMIO Access in #VE handler

To support an unmodified driver using MMIO, a TDVF may implement a Virtualization Exception (#VE) handler for an extended page table (EPT) violation. The #VE handler will use the guest physical address (GPA) as the MMIO address in TDCALL[TDG.VP.VMCALL]<#VE.RequestMMIO>.

In #VE handler, the TDVF should check if the SHARED bit is set in the GPA, and should reject this request if the SHARED bit is clear. Otherwise, the VMM could remove an accepted page and TDH.MEM.PAGE.AUG the same page to put the page in PENDING state. Then the VMM could use

TDCALL[TDG.VP.VMCALL]<#VE.RequestMMIO> to inject the malicious data to the private page.

2.6.4.3 TD ATTRIBUTES.SEPT_VE_DISABLE

Ideally, the VMM should set SEPT_VE_DISABLE in the TD ATTRIBUTES to prevent an EPT violation to #VE caused by guest TD access of PENDING pages.

The TDVF early boot code should read the TD ATTRIBUTES via TDCALL[TDG.VP.INFO] to ensure the VMM sets the SEPT_VE_DISABLE bit. Otherwise, there is risk in the #VE handler for this EPT violation.

2.6.4.4 TDCS.NOTIFY_ENABLES

TDX module may raise a #VE as a notification mechanism when it detects excessive Secure EPT violations raised by the same TD instruction (zero-step attack is suspected by TDX module). This is only done if bit 0 of TDCS.NOTIFY_ENABLES field is set.

TDCS.NOTIFY_ENABLES is 0 by default. The TDVF should not set the TDCS.NOTIFY_ENABLES via TDCALL[TDG.VM.WR]. Otherwise, there is risk in the #VE handler for this EPT violation.

3 TDVF Binary Image

This chapter describes the TDVF binary image format.

3.1 Boot Firmware Volume (BFV)

The TDVF includes one Firmware Volume (FV) known as the Boot Firmware Volume. The FV format is defined in the UEFI Platform Initialization (PI) specification.

The Boot Firmware Volume includes all TDVF components required during boot.

The file system GUID must be **EFI_FIRMWARE_FILE_SYSTEM2_GUID** or **EFI_FIRMWARE_FILE_SYSTEM3_GUID**, which is defined in PI specification.

1) **TdResetVector** – this component provides the entrypoint for TDVF, switch to long mode, and jumps to the DxeIpl. The FFS GUID must be EFI_FFS_VOLUME_TOP_FILE_GUID, which is defined in PI specification.

2) **TdDxeIpl** – This component prepares the required parameter for DxeCore and jumps to DxeCore.

3) **DxeCore** – This is standard DxeCore, used in standard UEFI firmware. It dispatches all DXE modules.

4) DXE Modules – These are TDVF-specific modules, to initialize the TDVF environment and launch the OS loader.

The BFV may include an initial static page table to assist the ResetVector switch from 32-bit mode to 64-bit mode.

3.2 Configuration Firmware Volume (CFV)

TDVF may also include a configuration firmware volume (CFV) that is separated from the boot firmware volume. The reason to do this is because the CFV is measured in RTMR, while the boot FV is measured in MRTD.

Configuration Firmware Volume includes all the provisioned data. This region is read only. One possible usage is to provide UEFI Secure Boot Variable content in this region, such as PK, KEK, db, dbx.

This region may include additional configuration variables.

The file system GUID must be **EFI_SYSTEM_NV_DATA_FV_GUID**, which is defined in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/SystemNvDataGuid.h.

The variable storage header must be VARIABLE_STORE_HEADER, and the variable header must be VARIABLE_HEADER. Both are defined in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/VariableFormat.h.

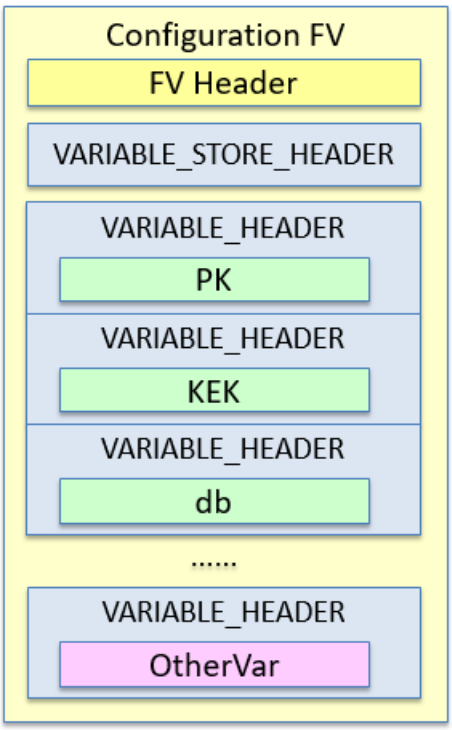

Figure 3-1: TDVF Configuration Firmware Volume

4 TD Launch

This chapter describes how a VMM passes control to TDVF.

4.1 TDVF initialization

Because TDVF relies on Intel TDX module, initialization flow is different from the typical firmware used for VM support (e.g. Open Virtual Machine Firmware (OVMF)).

4.1.1 VCPU Init State

When Intel TDX module launches TDVF, the virtual CPU (VCPU) executes in 32-bit protected mode with flat descriptor paging disabled. See **[TDX-CPU]** for details on the VCPU init state.

4.1.2 System Information

When the VMM calls Intel TDX module to launch TDVF, the VMM must build system information and pass it to TDVF as part of a TD HOB in RCX and R8 (the system memory location) if the TD_HOB section is present in the TDVF metadata area. If the TD_HOB section in the TDVF metadata area is absent, then the PermMem (permanent memory) section must be present and VMM must follow the memory requirement in the permanent memory section.

Please refer to 'TD Memory Management' for details on memory information reporting.

4.1.3 Long Mode Transition

The 32-bit TDVF init code will set up paging and switch to long mode. Because TDCALL is only valid in long mode, the 32-bit TDVF init code cannot invoke TDCALL to convert private memory regions, and 32-bit TDVF init code cannot access any permanent memory.

The 32-bit TDVF init code may refer to an initial temporary page table inside of the TDVF flash image (ROM page table), which is created at build time. Because the page table is not updatable, the page table Access Bit and Dirty Bit must be set to 1. The ROM page table must be in the Boot Firmware Volume if it is implemented.

Once TDVF switches to long mode it will obtain permanent memory information from the resource description HOB, allocate private memory, and create the final page table in private memory.

The TDVF should refer to GPAW (**RBX[0:6]**) to decide how much memory can be covered by the page table. If the GPAW is 48bit, then TDVF should set up 4 level paging. If the GPAW is 52bit, the TDVF should set up 5 level paging.

4.1.4 Setup stack to call C function

The 32-bit TDVF init code runs on the flash. Because it cannot access any permanent memory, the code cannot use stack, which is required by C language.

After TDVF switches to the long mode and issues TDCALL to convert the shared memory to private memory, the TDVF can set up stack in the private memory and call C function.

4.1.5 Switch to UEFI environment

The purpose of PEI is to detect and initialize memory. Since TDVF already knows the memory information via the HOB, the TDVF may skip PEI phase for information collection, and jump directly to DXE. If more HOB entry is required, the TDVF may allocate a new HOB from permanent memory and pass it to DXE.

The whole boot flow is shown below:

Stage 1: TD Init Code - 32-bit protected mode (stackless)

1) Use the TDVF flash image for ROM page table.
2) Switch to long mode

Stage 2: TD Init Code - 64-bit long mode (stackless)

1) Parse the TD Hob to get the memory location.
2) Set up initial private page, now we can use memory.
3) Set up temp stack from private page
4) Jump to C-code.

Stage 3: TD Init Code - 64-bit long mode (C-code)

1) Setup TD Initial Heap from private page.
2) Create the DXE Hob in the heap, based upon TD Hob.
3) Setup final page table in the heap.
4) Set up final stack in the heap.
5) Relocate DXE core in the heap.
6) Jump to DXE

Stage 4: UEFI environment

1) Dispatch drivers.
2) Wake up APs
3) Setup ACPI table
4) Prepare TD measurement
5) Prepare memory map
6) Start Console and storage Device
7) Invoke OS loader

Stage 5: OS environment

1) Init OS
2) Wake up APs.

The following figure shows the general TDVF flow.

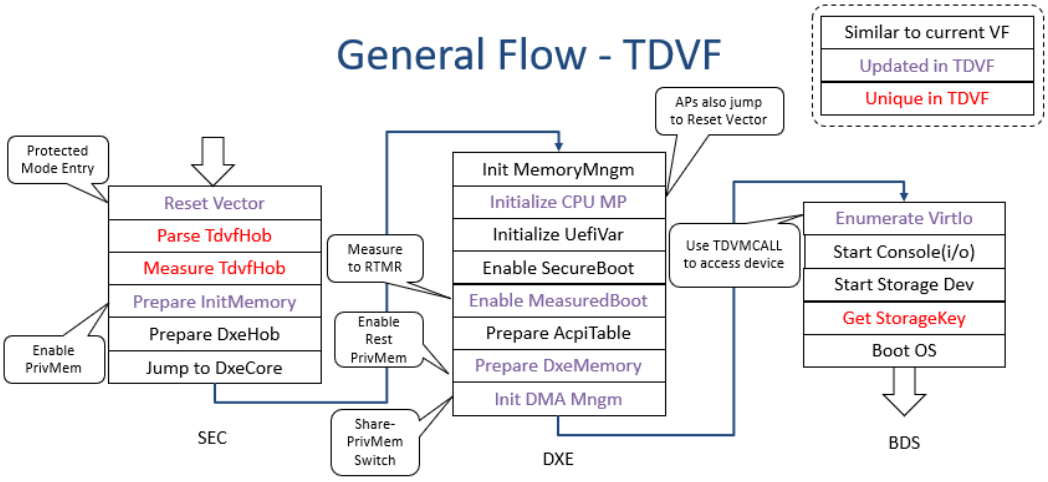

Figure 4-1: TDVF General Flow

4.2 TD Hand-Off Block (HOB)

The TD HOB list is used to pass the information from VMM to TDVF. The HOB format is defined in PI specification.

The TD HOB must include PHIT HOB, Resource Descriptor HOB. Other HOBs are optional.

The TDVF must create its own DXE HOB based upon TD HOB and pass the DXE HOB to DXE Core. The DXE HOB requirements are described in the UEFI PI specification.

4.2.1 PHIT HOB

The TD HOB must include PHIT HOB as the first HOB. `EfiMemoryTop`, `EfiMemoryBottom`, `EfiFreeMemoryTop`, and `EfiFreeMemoryBottom` shall be zero.

4.2.2 Resource Description HOB

The TD HOB must include at least one Resource Description HOB to declare the physical memory resource.

Any DRAM reported here should be **accepted** by TDVF, except the Temporary memory and TD HOB regions, which are declared in the TD metadata (see section 11). The resource HOB may optionally report the MMIO and IO regions based on the guest hardware provided by the VMM.

4.2.3 CPU HOB

The CPU HOB is optional; if it is included the TDVF must ignore it and create its own CPU HOB for DXE. This CPU HOB shall have **SizeOfMemorySpace** equal to GPAW (48 or 52) and **SizeOfIoSpace** (0 or 16) based on the IO reported by the Resource HOB.

4.2.4 GUID Extension HOB

The TD HOB may include the GUID extension HOB to describe the TD Feature, which is VMM or TDVF specific. Any GUID extension HOB in TD HOB must be passed to DXE HOB.

4.3 TDVF AP handling

4.3.1 AP Init State

The AP init state is exactly same as the BSP init state.

The **VCPU_INDEX** is reported by **INIT_STATE.RSI** or **TD_INFO.R9[0:31]**. It is the starting from 0 and allocated sequentially on each successful **TDINITVP**.

The **NUM_VCPUS** is reported by **TD_INFO.R8[0:31]**. It is the Number of Virtual CPUs that are usable, i.e. either active or ready. The TDVF need use this number to determine how many CPUs will join.

The **MAX_VCPUS** is reported by **TD_INFO.R8[32:63]**. It is TD's maximum number of Virtual CPUs. This value should be ignored by the TDVF in this version. It may be used for other purpose in future version such as later-add.

Intel TDX module will start the VCPU with **VCPU_INDEX** from **0** to **(NUM_VCPUS – 1)**. As such, the TDVF can treat the BSP as the CPU with **VCPU_INDEX 0**. However, the TDVF cannot assume that the CPU with **VCPU_INDEX 0** is the first one to launch. The TDVF needs to rendezvous in early initialization code, let the BSP execute the main boot flow and let APs execute in the wait loop.

4.3.2 AP Information Reporting from VMM to TDVF

In TDX, there is no INIT/SIPI protocol. The expectation is that VMM need launch all VCPU to the TDVF entrypoint. After reset, all CPUs run the same initialization code. TDVF will do the BSP selection.

There might be several ways for BSP selection. Here the TDVF may rely on the information from Intel TDX module, but the TDVF must not rely on the information from VMM.

1) All CPUs try to set a global flag. The first CPU set the flag is elected as the BSP and does the rest of BSP work. The rest CPUs just wait for the release signal from the BSP, then does the rest of AP work. (Do not rely on the information provided by Intel TDX module)

2) CPU with VCPU_INDEX 0 is BSP. CPUs with non-0 VCPU_INDEX are APs. (Rely on the information provided by Intel TDX module)

The pseudo code in TDVF is below:

```
==================
VOID
BspSelection (
  VOID
  )
{
  if (VCPU_INDEX == 0) {
    BspInit()
  } else {
    ApInit()
  }
}

VOID
BspInit (
  VOID
  )
{
  // Set up page table
  // Jump to 64-bit mode.
  // Set up AP MPWK mailbox.
  // Wakeup AP for init rendezvous
  // Do rest of initialization
  // Wakeup AP to perform required function
  // Jump to OS
}

VOID
ApInit (
  VOID
  )
{
  // Wait for BSP init notification
  // Jump to the 64-bit mode setup by BSP (page table, etc.)
@Wait:
  // Wait in MPWK mailbox
  // Do the task assigned in MPWK mailbox
  // jump @Wait:
}
==================
```

4.3.3 AP initialization in TDVF

In TDVF, there is no need to do normal CPU initialization such as configure MTRR, or patch Microcode. The AP is just in a wait-for-procedure state.

4.3.4 AP information reporting from TDVF to OS

In TDVF, the CPU information is reported via ACPI MADT table. The MADT need report the existing APIC ID and processor UID in ASL and enabled flags.

The existing APIC ID may be got from the TD_VCPU_INFO_HOB or MAX_VCPUS.

The ACPI driver need assign processor UID and match them in ASL code and MADT.

The enabled flag may be reported by TD_VCPU_INFO_HOB with PROCESSOR_ENABLED_BIT, or NUM_VCPUS or confirmed by the AP itself.

4.3.5 AP initialization in OS

For the system that does not support INIT-SIPI-SIPI, the platform firmware publishes an ACPI MADT MPWK STRUCT in the MADT ACPI table. Please refer to the TDX Guest Hypervisor Communication Interface document for the detailed data structure.

The firmware pseudo code is shown below:

```
==================
VOID
BspInitMailBox (
  VOID
  )
{
  MailBox->ApicId = ACPI_MPWK_APICID_INVALID;
  MailBox->WakeupVector = 0;
  MailBox->Command = AcpiMpwkCommandNoop;
}

VOID
ApWaitForWakeup (
  VOID
  )
{
  while (TRUE) {
    if (MonitorSupported()) {
      Monitor (MailBox);
      Mwait (Extension, Hint);
    }
    else if (UMonitorSupported()) {
      UMonitor (MailBox);
      UMwait (Extension, Hint);
    }
    else {
      Pause ();
    }
    //
    // Wait for wakeup
    //
    if (MailBox->Command == AcpiMpwkCommandNoop) {
      continue;
    }

    //
    // Check if for me
    //
    if (!ApIsMyMessage (MailBox->ApicId) {
      continue;
```

```
    }

    //
    // Dispatch command
    //
    switch (MailBox->Command) {
    case AcpiMpwkCommandWakeup:
      //
      // 64-bit vector
      //
      UINT64 WakeupVector = MailBox->WakeupVector;
      //
      // Ack MailBox
      //
      ReadWriteBarrier();
      MailBox->Command = AcpiMpwkCommandNoop;
      ReadWriteBarrier();

      Jump64 (WakeupVector)
      //
      // Never returns
      //
      CpuDeadLoop();
      break;
    default:
      break;
    }
  }
}

BOOLEAN
ApIsMyMessage (
  UINT32 ApicId
  )
{
  if (ApicId == MyApicId()) {
    return TRUE;
  }
  return FALSE;
}

==================
```

The OS pseudo code is shown below:

```
==================

VOID
BspWakeupThis (
  UINT32 ApicId
  )
{
  ReadWriteBarrier();
  MailBox->Command = AcpiMpwkCommandNoop;
  ReadWriteBarrier();
  //
  // Fill the mailbox
```

```
  //
  MailBox->ApicId = ApicId;
  MailBox->WakeupVector = ApWakeupAddress();

  ReadWriteBarrier();
  MailBox->Command = AcpiMpwkCommandWakeup;
  ReadWriteBarrier();
  //
  // Wait to join
  //
  Command = AcpiMpwkCommandWakeup;
  while ((Command != AcpiMpwkCommandNoop) && (!IsTimeout())) {
    ReadWriteBarrier();
    Command = MailBox->Command;
    ReadWriteBarrier();
    CpuPause();
  }

  if (GetJoinedCpuApicId() != ApicId) {
    // Something Wrong
  } else {
    // Good, new AP is here
  }
}

==================
```

§

5 TDVF UEFI Secure Boot Support

This chapter describes TDVF support for UEFI Secure Boot.

5.1 Provisioning UEFI Secure Boot

The TDVF creator should be responsible for provisioning the UEFI Secure Boot variable as the CFV.

The initialization code may build a variable hob and pass the information to variable driver. The GUID hob **gEfiAuthenticatedVariableGuid** is defined in https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Guid/VariableFormat.h.

5.2 Variable Driver

The TDVF variable driver uses the emulation variable driver in EDKII. This emulation driver does not support non-volatile storage and uses variable storage in RAM.

During variable initialization, the variable driver consults the variable GUID HOB data and initializes the variable storage region in RAM.

§

6 TDVF ACPI Support

This section describes TDVF ACPI Support.

Please refer to the ACPI specification for detailed information on what ACPI tables should be reported to OS.

6.1 Source of ACPI Tables

There are different ways to create ACPI table. For example:

1) The ACPI table may be created at build time, as the part of Boot Firmware Volume. It is measured automatically.

2) The ACPI table may be input at launch time, as the part of TD Hob or Configuration Firmware Volume. It is measured automatically.

3) The ACPI table may be input via a hypervisor-specific configuration interface. If this method is chosen, the consumer in TDVF must explicitly measure the raw ACPI table data to RTMR register. (See chapter 8, TD measurement).

6.2 ACPI Support

- ACPI S5 should be supported.

6.3 FADT

FADT should be configured for 'no ACPI Hardware' mode.

Due to the lack of SMM support, any SMI command field in FADT must be 0.

6.4 DSDT

The DSDT may report PCI or IO device based upon the device emulated by the VMM.

6.5 FACS

No ACPI S3 support is required.

6.6 MADT

Please refer to 'TDVF AP handling' (Section 4.3) for information on AP handling.

§

7 TD Memory Management

This chapter describes the memory management.

7.1 Memory Type

There are four defined types of TD memory:

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 I/O (MMIO)** – Shared memory accessed by TDVF via TDCALL [TDG.VP.VMCALL] <#VE.RequestMMIO>

The private memory type should be used by default.

Unaccepted memory is a special type of private memory. The TDVF must invoke TDCALL [TDG.MEM.PAGE.ACCEPT] the unaccepted memory before use it.

The shared memory can be converted from private memory. It is used for information pass from VMM or for IO buffer including Direct Memory Access (DMA). It must not be used for page table or executable. The private page is the default memory type in a TD for confidentiality and integrity. It must NOT be used for IO buffer.

The MMIO is a special shared memory. It can only be accessed via TDCALL [TDG.VP.VMCALL] <#VE.RequestMMIO>. It cannot be accessed via direct memory read or write.

7.1.1 Private Memory Indicator in Guest Page Table.

The guest indicates if a page is shared using the Guest Physical Address (GPA) Shared (S) bit. If the GPA Width (GPAW) is 48, the S-bit is bit-47. If the GPAW is 52, the S-bit is bit-51.

7.2 Initial State from VMM

The memory map information is passed from VMM to TDVF, via resource description Hob. The following figure shows a sample memory layout.

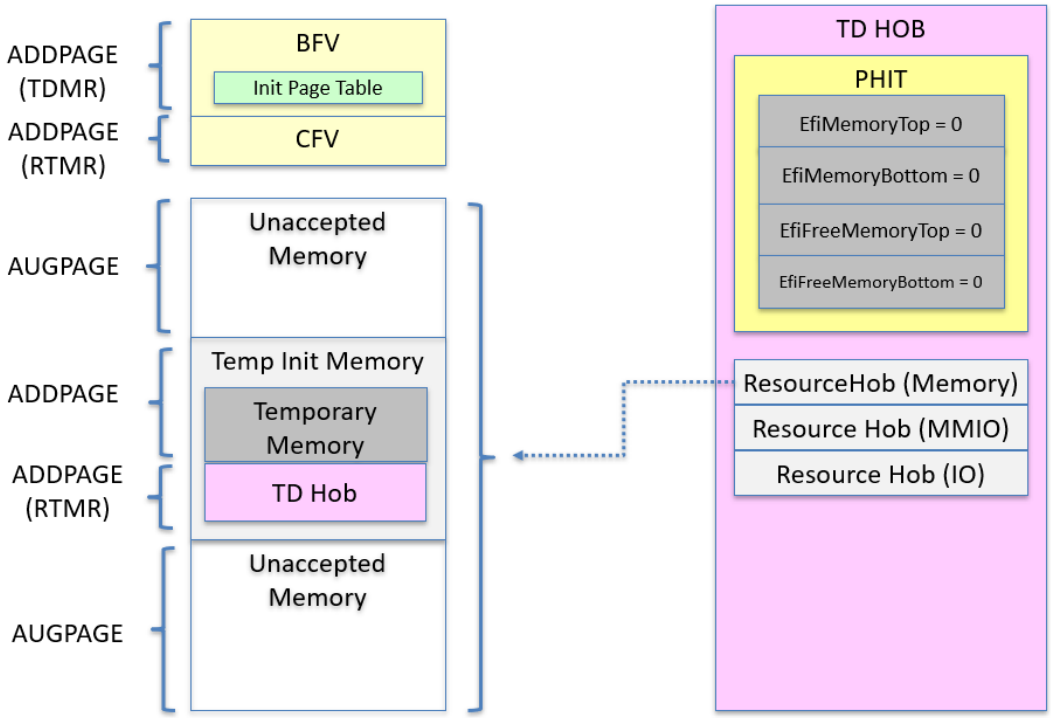

Figure 7-1: TD Hob and Initial Memory Layout

The VMM shall copy the BFV and CFV from the TDVF binary image, then fill TD_HOB to describe the memory layout if the TD_HOB is present in the TD metadata. If the TD_HOB is input from the VMM, the TD_HOB shall be measured by TDVF as the evidence of the initial TD memory configuration at the TD launch time.

7.2.1 Memory Type in TD Resource HOB

There are different types of memory in TD Hob, described in Table 7-1.

Table 7-1: Memory Type in TD Resource HOB

Memory Type	Report from VMM	VMM Action	TDVF Action
Private Memory	Optional, because TDVF can get the information from TD metadata directly. If it is reported, below format should be used TD Hob - Resource Hob Type: `EFI_RESOURCE_SYSTEM_MEMORY` Attributes: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED | EFI_RESOURCE_ATTRIBUTE_ENCRYPTED`	SEAMCALL [TDH.MEM.PAGE.ADD]	N/A Use directly

Unaccepted Memory	TD Hob - Resource Hob Type: `EFI_RESOURCE_MEMORY_UNACCEPTED` Attributes: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED`	SEAMCALL [TDH.MEM.PAGE.AUG]	TDCALL [TDG.MEM.PAGE.ACCEPT]
Memory-Mapped I/O (MMIO)	TD Hob - Resource Hob Type: `EFI_RESOURCE_MEMORY_MAPPED_IO` Attribute: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE`	N/A	Use directly via TDCALL [TDG.VP.VMCALL] <#VE.RequestMMIO>

The TDVF may indicate a small chunk of temporary initialized memory (added by SEAMCALL [TDH.MEM.PAGE.ADD]) for temporary usage before the TDVF accepts the unaccepted memory (added by SEAMCALL [TDH.MEM.PAGE.AUG]). The temporary initialize memory size can be small to support initial TDVF code finishing the memory initialization.

Any physical memory reported in TD Hob should be ACCEPTED by TDVF, except the Temporary memory and TD HOB regions, which are declared in the TD metadata. TDVF must consult the TD metadata to not accept any ADDED memory region, such as BFV, CFV, TD Hob, and Temporary memory. TDVF must only accept the unaccepted memory reported by VMM. The summary of the TDVF memory state from VMM is shown in Table 7-2.

Table 7-2: TDVF memory state from VMM

Component	Provider	Data	Report	VMM Action	TDVF Action	Measurement
Static Firmware Code (BFV, Page table)	TDVF (Tenant)	Initialized	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD] SEAMCALL [TDH.MR.EXTEND] SEAMCALL [TDH.MR.FINALIZE] (called after all SEAMCALL [TDH.MEM.PAGE.ADD])	N/A	MRTD
Static Firmware Configuration (CFV, UEFI variable)	TDVF (Tenant)	Initialized	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD]	TDCALL [TDG.MR.RTMR.EXTEND]	RTMR[0]
Dynamic Runtime Configuration (TD Hob)	VMM (CSP)	Initialized	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD]	TDCALL [TDG.MR.RTMR.EXTEND]	RTMR[0]

Temporary Initialized TD Memory	VMM (CSP)	0	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD]	N/A	N/A
Unaccepted TD Memory	VMM (CSP)	0	TD Hob - Resource Hob	SEAMCALL [TDH.MEM.PAGE.AUG]	TDCALL [TDG.MEM.PAGE.ACCEPT]	N/A
MMIO	VMM (CSP)	Initialized	TD Hob - Resource Hob	N/A	N/A	N/A
Disk	TD Guest (Tenant)	Not loaded	N/A	Virtual Disk Access	TDCALL [TDG.MR.RTMR.EXTEND] for the content in the disk, but not the full disk.	RTMR[1] for OS loader, kernel, initrd, boot parameter. NOTE: RTMR[2] is reserved for OS application measurement by OS kernel. RTMR[3] is reserved only for special usage.

7.3 Memory Information for DXE Core

The initial TDVF code needs to build a DXE HOB based on the TD HOB to pass the memory information to DXE Core. Figure 7-2 shows the DXE HOB and Runtime Memory Layout.

The flow of DXE HOB creation is below:

1) TDVF only uses the temporary initialized memory for temporary stack and temporary heap.

2) TDVF scans the TD HOB to get the memory layout information.

3) TDVF initializes permanent memory (by calling TDCALL [TDG.MEM.PAGE.ACCEPT]) from the top of the usable memory below 4GiB. The size of initial permanent memory can be small to support initial DXE core finishing the initialization and running memory test for the rest of permanent memory.

4) TDVF sets up heap from the top of the usable memory below 4GiB. The page table, stack, or DXE Core memory can be allocated there.

5) TDVF constructs the DXE_HOB in the permanent memory.

    a. The PHIT points to the location in the permanent memory.

    b. The FV Hob is created based upon TD Metadata. (See chapter 11)

    c. The CPU Hob is created based upon memory GPAW bit (48 or 52 SizeOfMemorySpace) and IO resource reporting (0 or 16 SizeOfIoSpace)

    d. The Resource Hob is created based upon the memory initialization state. The TDVF may optionally allocate shared memory. If this is the case, the shared memory should be reported as a stand-alone HOB entry without `EFI_RESOURCE_ATTRIBUTE_ENCRYPTED`. (See below section)

    e. The Memory Allocation Hob is created to describe the allocated memory in heap, such as permanent page table, stack, or DxeCore module.

6) The TDVF reclaims the temporary memory.

7) The TDVF handles the MMIO reported by TD HOB.

    a. If the MMIO region is for fixed resource such as APIC and HPET, the MMIO region should be reported in the DXE HOB.

    b. If the MMIO region is for dynamic resource allocation and will be managed by a dedicated DXE driver, then the MMIO region might **not** be reported in the DXE HOB. (Such as for an MMIO space for PCI MMIO BAR allocation by the PCI host bridge driver.) If the MMIO region is reported, then the HOB resource attribute `EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE` must be used, because PCI host bridge driver will try to add this region with GCD attribute `EFI_MEMORY_UC`.

8) The TDVF handles the Port IO reported by TD HOB.

    a. If the port IO region is for fixed resource such as 8254 and 8259, the port IO region should be reported in the DXE HOB.

    b. If the port IO region is for dynamic resource allocation and will be managed by a dedicated DXE driver, then IO region might **not** be reported in the DXE HOB. (For example, for an IO space for PCI IO BAR allocation by the PCI host bridge driver.)

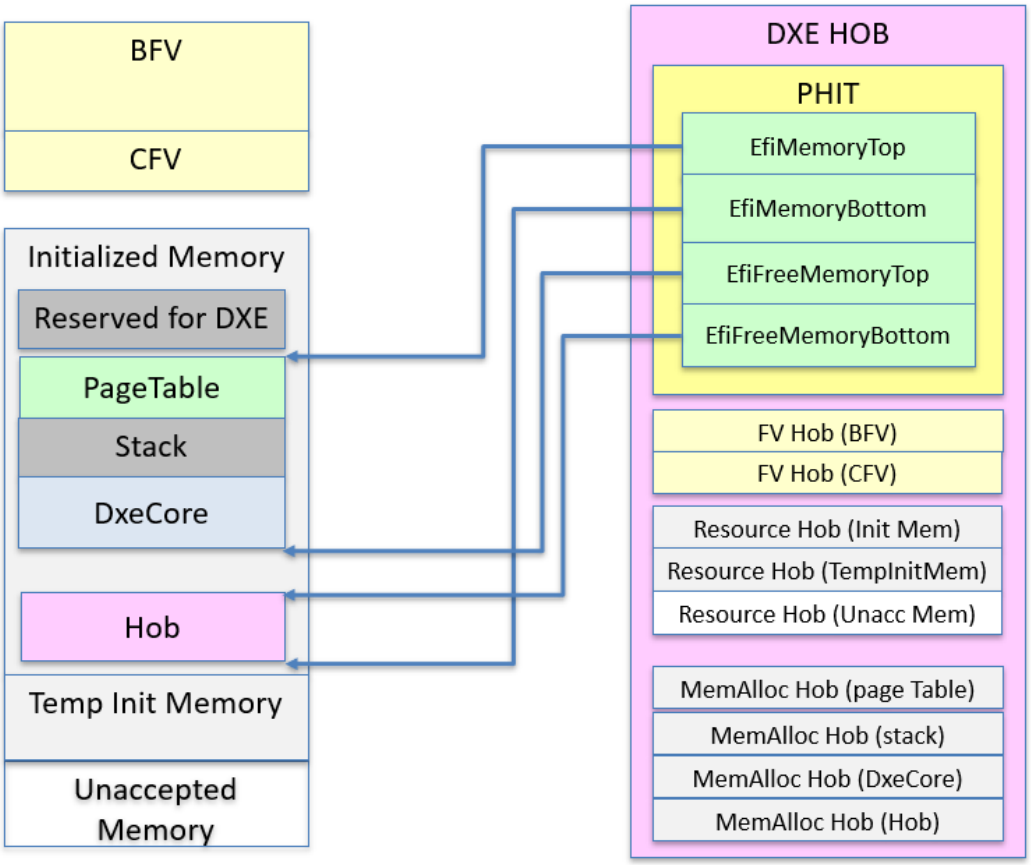

Figure 7-2: DXE HOB and Runtime Memory Layout

7.3.1 Memory Type in DXE Resource HOB

There are different types of memory in DXE Hob. See table 7-3.

Table 7-3: Memory Type in DXE resource HOB

Memory Type	Report from TDVF initial code	VMM/TDVF initial code Action	TDVF Action
Private Memory	DXE Hob - Resource Hob Type: `EFI_RESOURCE_SYSTEM_MEMORY` Attributes: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED | EFI_RESOURCE_ATTRIBUTE_ENCRYPTED`	SEAMCALL [TDH.MEM.PAGE.ADD] / TDCALL [TDG.MEM.PAGE.ACCEPT]	Use directly
Shared Memory (optional)	DXE Hob - Resource Hob Type: `EFI_RESOURCE_SYSTEM_MEMORY` Attributes: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED`	SEAMCALL [TDH.MEM.PAGE.ADD] / TDCALL [TDG.MEM.PAGE.ACCEPT]	Use directly

Unaccepted Memory	DXE Hob - Resource Hob Type: `EFI_RESOURCE_MEMORY_UNACCEPTED` Attributes: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TEST`	SEAMCALL [TDH.MEM.PAGE.AUG]	TDCALL [TDG.MEM.PAGE.ACCEPT]
Memory-Mapped I/O (MMIO)	DXE Hob - Resource Hob Type: `EFI_RESOURCE_MEMORY_MAPPED_IO` Attribute: `EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_UNCACHEABLE`	N/A	Use directly via TDCALL [TDG.VP.VMCALL] <#VE.RequestMMIO>

7.4 Memory Map to TD-OS

If a memory region is private memory, the final UEFI memory map shall report the region with normal UEFI memory type.

If a memory region is shared memory, the final UEFI memory map shall report the region with normal UEFI memory type. It is converted by the IOMMU driver to private memory automatically at **`ExitBootServices`** event.

If a memory region is unaccepted memory and requires TDCALL [TDG.MEM.PAGE.ACCEPT]in the TD guest OS, then the final UEFI memory map shall report this region in **`EfiUnacceptedMemoryType`**. The OS need TDCALL [TDG.MEM.PAGE.ACCEPT] before use it.

If a memory region is MMIO, it can only be accessed via TDCALL [TDG.VP.VMCALL] <#VE.RequestMMIO>. It cannot be accessed via direct memory read or write. There is no need to report this region in UEFI memory map because no RUNTIME attribute is required. The full MMIO regions should be reported in ACPI ASL code via memory resource descriptors.

For non-UEFI system, the memory map is reported via E820 table. The private memory is reported as normal E820 memory type. The unaccepted memory is reported as **`AddressRangeUnaccepted`** type.

The TDVF need report the memory map information to OS. Please refer to the TDX Guest Hypervisor Communication Interface document for the detailed information.

7.5 Convert Shared to Private

The TDVF need convert shared memory to private memory in late memory initialization or to reclaim virtual device IO buffer or hypervisor communication buffer.

The TDVF must take the following steps to convert shared memory to private memory:

- Guest removes GPA from shared space. Clear S-bit in page table.
- TDCALL [TDG.VP.VMCALL] <MAPGPA>
- TDCALL [TDG.MEM.PAGE.ACCEPT]

This step can be done by an IOMMU protocol **FreePages()/Unmap()** if this is for the virtual device IO buffer.

The IOMMU protocol definition can be found at https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Include/Protocol/IoMmu.h

7.6 Convert Private to Shared

The TDVF needs to convert private memory to shared memory for hypervisor communication such as TDCALL [TDG.VP.VMCALL] or virtual device IO buffer.

The TDVF should take the following steps to convert shared memory to private memory:

- Guest adds GPA to the shared space. Set S-bit in page table.
- TDCALL [TDG.VP.VMCALL] <MAPGPA>

This step can be done by an IOMMU protocol **AllocatePages()/Map()** if this is for the virtual device IO buffer.

7.7 Memory State Transition

Please see the figure below for the TDVF memory statue transition.

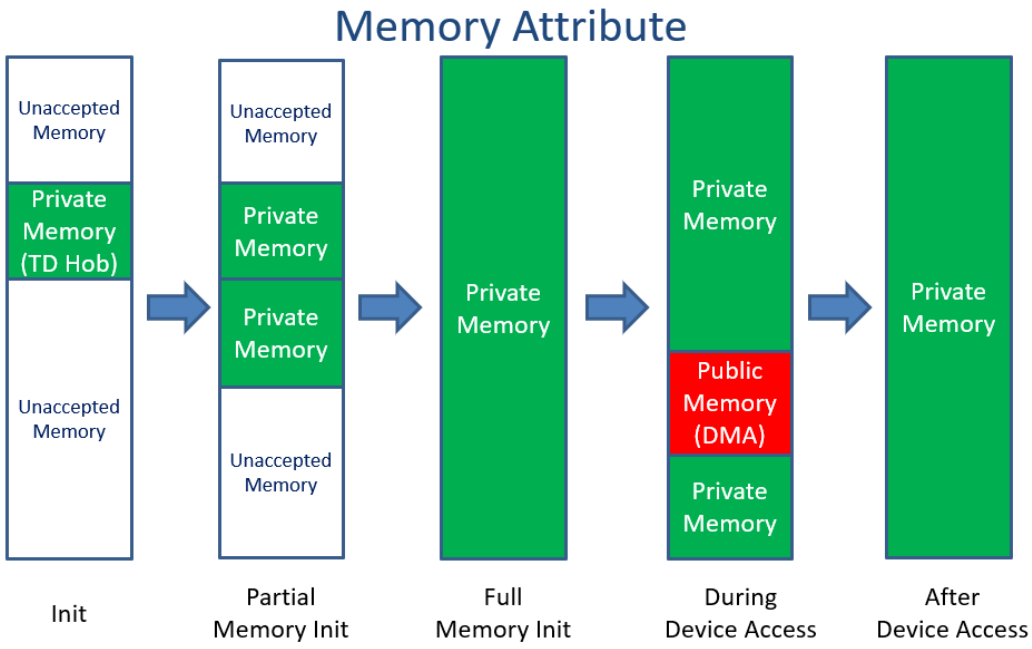

Figure 7-3: TDVF Memory State Transition

7.8 Optimization Consideration

The TDVF implementation may add all physical pages to private memory before transfer control to OS. This may increase the boot time, because the TDCALL [TDG.MEM.PAGE.ACCEPT] is time-consuming. As such, the TDVF may choose to add part of memory to be private memory and boot to OS. Then the TD-OS can convert the rest page to be private. The TDVF may do below optimization.

7.8.1 Partial Memory Initialization in Pre-UEFI

When the TDVF MemoryInit module initializes the memory - convert all memory to be accepted private memory, the MemoryInit module need mark the resource description HOB to be `EFI_RESOURCE_SYSTEM_MEMORY` with `(EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED)`.

There is no need to initialize all system memory in the early phase, where it is the single thread environment. The MemoryInit module just need initialize enough memory to launch the DXE Core. The MemoryInit module need split the resource description HOB into 2 parts: the initialized private or shared memory `EFI_RESOURCE_SYSTEM_MEMORY` with `(EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED)` and the unaccepted memory `EFI_RESOURCE_MEMORY_UNACCEPTED` with `(EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED)`.

For example, the MemoryInit may just need to initialize 256M memory for DXE core, even if there are 4G memory available.

7.8.2 Partial Memory Initialization in UEFI

Once the DXE Core is launched, it checks the resource description HOB and only uses the memory `EFI_RESOURCE_SYSTEM_MEMORY` with `(EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED)`. A MemoryTest module must go through the resource description HOB and do the late-initialization in DXE phase for the unaccepted memory `EFI_RESOURCE_MEMORY_UNACCEPTED` with `(EFI_RESOURCE_ATTRIBUTE_PRESENT | EFI_RESOURCE_ATTRIBUTE_INITIALIZED | EFI_RESOURCE_ATTRIBUTE_TESTED)`.

After the late initialization, the GCD type of the memory is converted from `EfiGcdMemoryTypeUnaccepted` to `EfiGcdMemoryTypeSystemMemory`

The MemoryTest module just needs to initialize enough accepted private memory for the UEFI environment, and launch the OS loader. The accepted private memory is

reported as normal memory type. The unaccepted memory is reported as `EfiUnacceptedMemoryType`. This allows OS to do late initialization.

For example, the UEFI environment may only need to initialize 4G memory for the OS loader if there is 256G memory available.

7.8.3 Parallelized Memory Initialization

The MemoryTest module is executed in the multi-processor environment.

The MemoryTest module may consider using multi-processor to parallelize the memory initialization process. The BSP may split the task and wake up all APs and let multiple APs do the memory initialization.

For example, if a system has 16 CPUs and 8G memory to be initialized, the BSP may wake up 15 APs to let each CPU initialize 512M memory.

7.8.4 Pre-allocating Virtual Device IO Buffer

The TDVF may use virtual device IO buffer in the deriver driver to support system boot, such as virt-io, file system driver. Because the virtual device IO buffer is shared memory, there is risk of thrashing which means the excessive swapping between shared memory and private memory.

To avoid such thrashing, the IOMMU driver may pre-allocate a big chunk of shared memory as the virtual device IO buffer in the driver entrypoint, and only convert it back to private in the `ExitBootServices` event. The other virtual device IO buffer allocation should be inside of the pre-allocated IO buffer. As such, the overhead of the virtual device IO buffer converting can be reduced.

§

8 TD Measurement

This chapter describes the measurement/attestation/quote.

8.1 Measurement Register Usage in TD

TDs have two types of measurement registers:

- **TD measurement register (MRTD):** Static measurement of the TD build process and the initial contents of the TD.
- **Runtime Extendable measurement register (RTMR):** An array of general-purpose measurement registers, available to the TD software for measuring additional logic and data loaded into the TD at runtime.

A system has 1 MRTD and 4 RTMR. The typical usage is shown below:
- MRTD is for the TDVF code (match PCR[0]).
- RTMR[0] is for the TDVF configuration (match PCR[1,7]).
- RTMR[1] is for the TD OS loader or kernel (match PCR[4,5]).
- RTMR[2] is for the OS application (match PCR[8~15]).
- RTMR[3] is reserved for special usage only.

Table 8-1 shows how to match the PCR used in the regular platform to the TD Measurement Register used in TDVF.

Table 8-1: TD Measurement-related Register

PCR Index	Typical Usage	TD Register	TD Reg Index	Event Log	Extended by	Checked by	Content
0	Firmware Code (BFV, including init page table)	**MRTD**	0	NO	VMM: TDCALL [TDH.MR.EXTEND]	Remote	VF code (BFV)
1	Firmware Data (CFV, TD Hob, ACPI Table)	**RTMR [0]**	1	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	Static Configuration (CFV), Dynamic Configuration (TD HOB, ACPI)
2	Option ROM code	**RTMR [1]**	2	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	PCI Option ROM, such as NIC.

3	Option ROM data	**RTMR [1]**	2	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	
4	OS loader code	**RTMR [1]**	2	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	OS loader, OS kernel, initrd.
5	Configuration (GPT, Boot Variable)	**RTMR [1]**	2	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	GPT, Boot Variable, Boot Parameter.
6	N/A	N/A		N/A	N/A	N/A	
7	Secure Boot Configuration	**RTMR [0]**	1	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	SecureBootConfig (in CFV)
8~15	TD OS APP measurement	**RTMR [2]**	3	-	TD OS: TDCALL [TDG.MR.RTMR.EXTEND]	-	TD OS App. Done by OS.

8.2 Fundamental Support

Intel TDX module shall measure TD BFV and extend to MRTD. MRTD can be used for attestation purpose. (Similar to the trust boot flow)

During runtime, TDVF shall measure TD CFV and TD Hob to RTMR[0] and the OS code/data and extend to RTMR[1]. This action is done via TDCALL [TDG.MR.RTMR.EXTEND] that allows dynamic measurement extensions (Dynamic data). RTMR can be used for attestation purpose. (Similar to the trust boot flow)

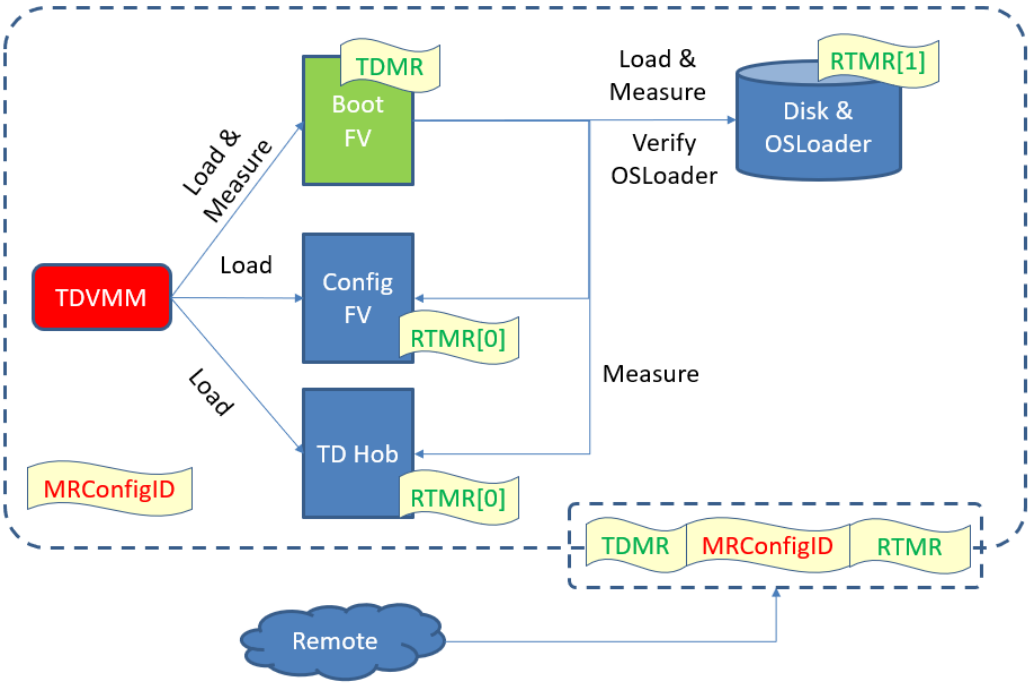

Figure 8-1: TDVF Measurement

Please see the Intel® TDX Guest-Hypervisor Communication Interface document for the detailed data structure.

The TDVF should expose EFI_CC_MEASURMENT_PROTOCOL and CCEL ACPI table. The original EFI_TCG2_PROTOCOL and TPM2 ACPI table should be used for virtual TPM only. See figure 8-2.

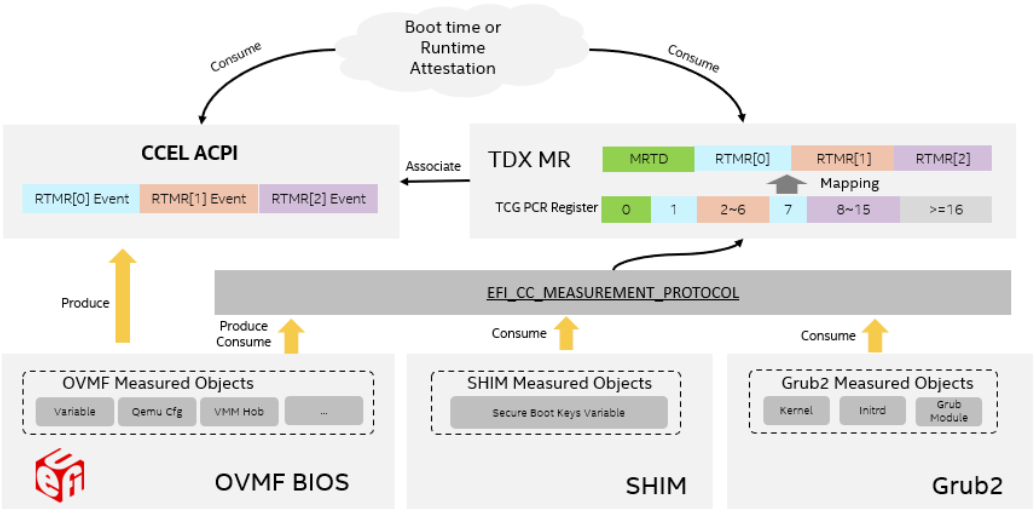

Figure 8-2: TDVF Measurement Interface

8.3 Virtual Firmware Configuration

The TDVF may support to consume different configuration data as the policy to control the code flow. Care must be taken that this is an attack surface, because the VMM is not trusted. Please use the following guidance:

- The TDVF should reduce configuration options to reduce the attack surface.
- Configuration data must be copied to private memory, measured, and verified before it is used.

8.3.1 Build-Time Configuration

The Build-Time configuration is the configuration firmware volume (CFV). It may include UEFI Secure Boot variables (PK, KEK, db, dbx).

The data will be exposed as UEFI variable.

8.3.2 Launch-Time Configuration

The Launch-Time configuration is the TD HOB passed from VMM. This hob may include the system configuration, such as the memory range, MMIO range, IO range, CPU number, etc.

The hypervisor vendor may add GUIDed Hob as the extension to provide hypervisor-specific information to the hypervisor-specific TDVF, such as ACPI tables and SMBIOS tables.

The data included in TD HOB must be accessible during the whole firmware boot time.

8.3.3 Runtime Configuration

Runtime configuration uses non-volatile UEFI variable data.

In order to simplify the design, TDVF does not support non-volatile variables.

If TDVF requires support for non-volatile variables in the future, the variable area must be measured into RTMR[0].

8.3.4 Hypervisor Specific Configuration Interface

Currently, some virtual firmware may use hypervisor-specific configuration interface to get the configuration information.

For example, OVMF uses **FW_CFG_IO_SELECTOR (0x510)** and **FW_CFG_IO_DATA** (0x511) to get configuration information from QEMU. Example information:

- "etc/system-states"
- "etc/table-loader"
- "etc/extra-pci-roots"
- "bootorder"
- "etc/boot-menu-wait"
- "etc/tpm/config"
- "etc/edk2/https/cacerts"
- "etc/msr_feature_control"
- "etc/e820"
- "opt/ovmf/X-PciMmio64Mb"
- "etc/reserved-memory-end"
- "etc/ramfb"
- "etc/smbios/smbios-tables"
- "etc/smi/supported-features"
- "etc/smi/requested-features"
- "etc/smi/features-ok"

If TDVF uses this method, configuration data must be measured into RTMR[0].

8.4 Attestation and Quote Support

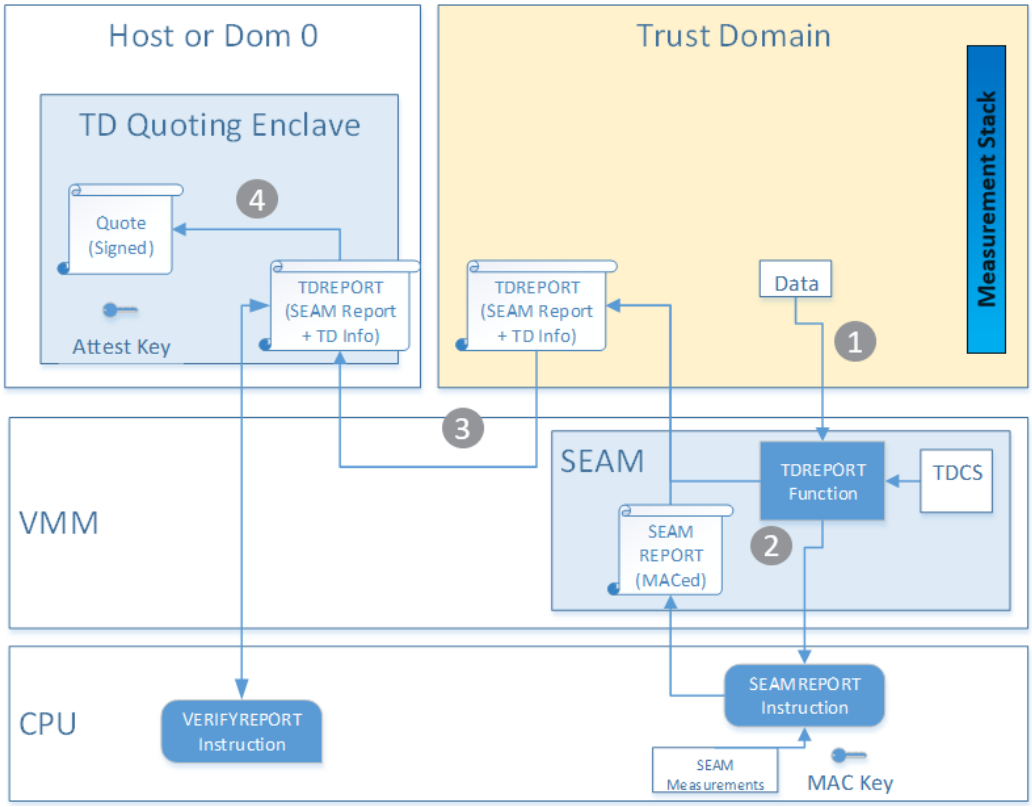

Figure 8-3: Attestation and Quote

1. Guest TD invokes TDCALL [TDG.MR.REPORT] API function.
2. Intel TDX module uses the SEAMOPS [SEAMREPORT] instruction to create a MAC TDREPORT_STRUCT with the Intel TDX module measurements from CPU and TD measurements from TDCS.
3. Guest TD uses TDCALL [TDG.VP.VMCALL] <GETQUOTE> to request TDREPORT_STRUCT be converted into Quote.
4. The TD Quoting enclave uses ENCLU[EVERIFYREPORT2] to verify the TDREPORT_STRUCT. This allows the Quoting Enclave to 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_STRCUT body with an ECDSA 384 signing key.

§

9 TDVF Device Support

TDVF device driver support must align with the TD-VMM capability. Microsoft Windows Hyper-V may require Vmbus. Linux KVM may require VirtIo.

9.1 Minimal Requirement

The following devices are enabled for a typical guest OS:

1) Debug Device (Serial Port)
2) Storage Device (Block device or SCSI)
3) Output Device (Graphic)
4) Input Device (Keyboard)
5) Network Device (LAN)

For a container use case, a simplified TDVF may jump directly to the OS kernel without enabling any devices.

9.2 VirtIo Requirement

VirtIo requires the following:

- virtio-pci (Enumeration over PCI)
- virtio-serial (serial output / debug)
- virtio-blk and/or virtio-scsi (Storage)
- virtio-gpu (Graphics)
- virtio-input (Keyboard)
- virtio-net and/or virtio-socket (Network)

9.3 Security Device

Support for virtio-rng is not required because the VMM is not trusted. TD must use the RDSEED or RDRAND instruction to obtain a random number.

9.4 HotPlug Device

The current TDX does not support CPU Hot Plug feature. As such, the TDVF does not support CPU Hot Plug. Memory Hot Plug can be supported by TDX architecture. The TDVF does not support the memory hot plug and only reports the memory map from VMM at TD launch time, including accepted memory and unaccepted memory. The TD OS may support memory hot plug at runtime as an optional feature.

1) CPU Hot Add – This is blocked by Intel TDX module. Intel TDX module injects all VCPU into the guest TD at one time. The MADT ACPI table holds and only holds the active CPUs reported by the Intel TDX module.

2) CPU Hot Remove – It can be used as a denial-of-service attack from the VMM. This feature is out of scope for TDX.

3) Memory Hot Add – VMM must use the standard defined process (such as ACPI) to notify the hot add event to TD OS. Otherwise, TD OS should ignore memory configuration changes.

4) Memory Hot Remove – VMM must use the standard defined process (such as ACPI) to notify the hot remove event to TD OS. Surprise removing can be used as a denial-of-service attack from the VMM. This feature is out of scope for TDX.

Although TDVF does not support Hot Plug, a VMM may report an ACPI table with Hot Plug support to access the guest OS. In order to mitigate this, any input from the VMM must be measured by OVMF into RMTD for attestation. Note that TDVF cannot verify ACPI table content. The guest OS should treat ACPI input as untrusted data and parse it carefully.

9.5 PCI Device Option ROM

UEFI x64 PCI Option ROMs (OROM) are supported. Before execution, the OROM must be measured into RTMR. Because UEFI Secure Boot is enabled, the PCI OROM must be signed with valid certificate against info enrolled into UEFI Secure Boot variables.

Legacy 16bit OROM and 32-bit UEFI OROM must be rejected and ignored.

§

10 Exception Handling

This chapter describes exceptions that may be injected by Intel TDX module.

10.1 Virtualization Exception (#VE)

The TDVF provides the default #VE exception handler. The handler is implementation-specific, like all other exception handlers.

The default handler may:

- Dump the exception reason via TDCALL [TDG.VP.VEINFO.GET] and the architecture state, including general-purpose registers, in debug mode for root-cause analysis
- Dead loop

Care must be taken that the debug output must NOT violate TDX restrictions. For example, IO port access is illegal in TD, so the Serial IO debug output must be modified with TDCALL [TDG.VP.VMCALL] <INSTRUCTION.IO>.

10.2 Instruction Conversion

The VE exception handler may also convert some of the forbidden instruction to the TDCALL [TDG.VP.VMCALL] <INSTRUCTION>. For example:

CPUID => TDCALL [TDG.VP.VMCALL] <Instruction.CPUID>

IO => TDCALL [TDG.VP.VMCALL] <Instruction.IO>

MMIO => TDCALL [TDG.VP.VMCALL] <#VE.RequestMMIO>RDMSR => TDCALL [TDG.VP.VMCALL] <Instruction.RDMSR>

WRMSR => TDCALL [TDG.VP.VMCALL] <Instruction.WRMSR>

Care must be taken that when the #VE handler should not use TDCALL[TDG.VP.VMCALL]<#VE.RequestMMIO> to access any private memory caused by the EPT violation. Please refer to Chapter 2 Security Consideration.

§

11 TDVF Metadata

This section describes TDVF Metadata. This metadata provides information to the VMM that is used to build a TD.

11.1 TDVF Metadata Location

The metadata is located at (TDVF end – 0x20) byte. It is a 4-bytes offset of the TDVF_DESCRIPTOR to the beginning of the TDVF image.

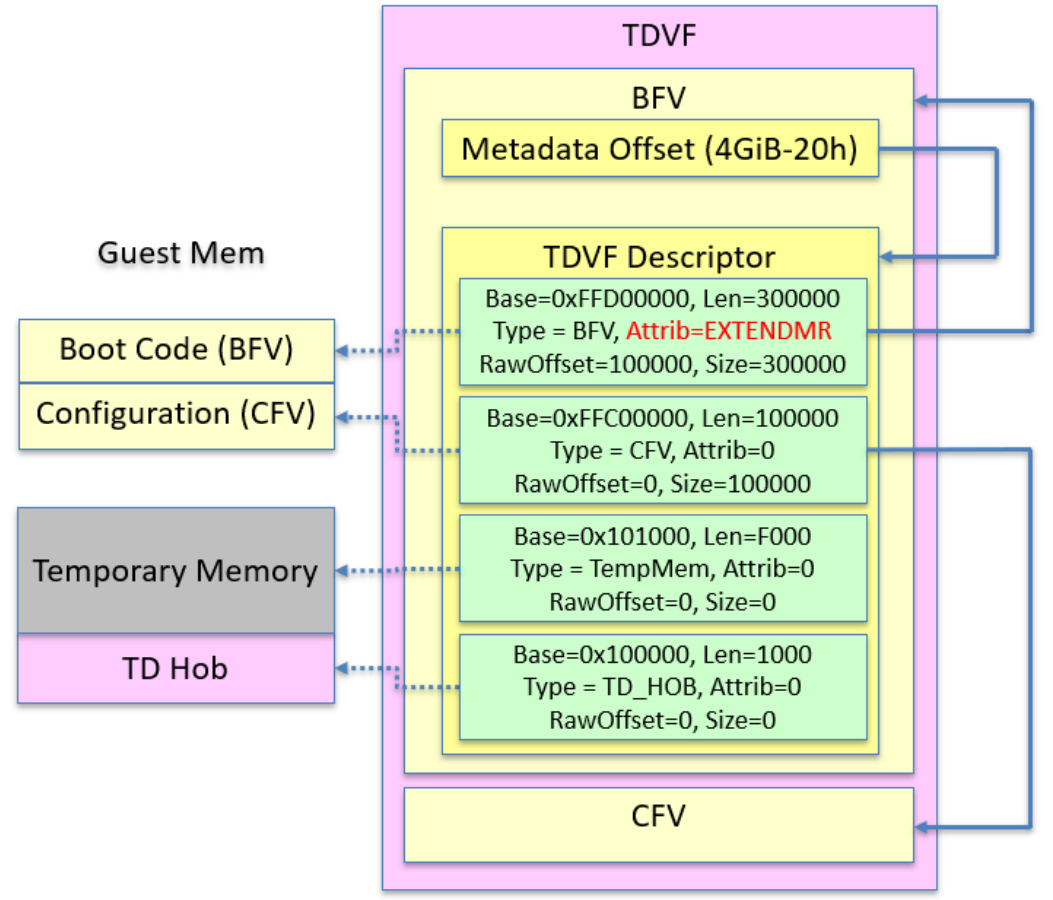

Figure 11-1: TDVF Metadata Layout

11.2 TDVF descriptor

The VMM refers to TDVF_DESCRIPTOR to set up memory for TDVF.

Table 11-1: TDVF_DESCRIPTOR definition

Field	Offset (Byte)	Type	Size (Byte)	Description
Signature	0	CHAR8[4]	4	"TDVF"
Length	4	UINT32	4	Size of the structure (d)
Version	8	UINT32	4	Version of the structure. It must be 1.
NumberOfSectionEntry	12	UINT32	4	Number of the section entry (n)
SectionEntries	16	TDVF_SECTION[n]	32*n	See table 13-2.

Table 11-2: TDVF_SECTION definition

Field	Offset (Byte)	Type	Size (Byte)	Description
DataOffset	0	UINT32	4	The offset to the raw section in the binary image.
RawDataSize	4	UINT32	4	The size of the raw section in the image. If it is zero, the VMM shall allocate zero memory from MemoryAddress to (MemoryAddress + MemoryDataSize). If it is zero, then the DataOffset shall also be zero.
MemoryAddress	8	UINT64	8	The guest physical address of the section loaded. It must be 4K aligned.
MemoryDataSize	16	UINT64	8	The size of the section loaded. It must be 4K aligned. It must be non-zero value. It must be not less than RawDataSize. If MemoryDataSize is greater than RawDataSize, the VMM shall fill zero up to the MemoryDataSize.
Type	24	UINT32	4	The type of the TDVF_SECTION. See table 13-3.
Attributes	28	UINT32	4	The attribute of the section. See table 13-4.

Table 11-3: TDVF_DESCTION.Attributes definition

Value	Name	Memory Type	VMM Action	Td-Shim Action	Measurement
0	BFV/TdShim	Private Memory	PAGE.ADD + MR.EXTEND	N/A	MRTD
1	CFV	Private Memory	PAGE.ADD	RTMR.EXTEND	RTMR[0]
2	TD_HOB	Private Memory	PAGE.ADD	RTMR.EXTEND	RTMR[0]
3	TempMem	Private Memory	PAGE.ADD	N/A	N/A
4	PermMem	Unaccepted Memory	PAGE.AUG	PAGE.ACCEPT	N/A
5	Payload	Private Memory	PAGE.ADD + MR.EXTEND*	RTMR.EXTEND*	MRTD (or) RTMR[1]
6	PayloadParam	Private Memory	PAGE.ADD	RTMR.EXTEND	RTMR[1]
7 ~ 0xFFFFFFFF	Reserved	N/A	N/A	N/A	N/A

Table 11-4: TDVF_DESCTION.Attributes definition

Bits	Name	Description
0	MR.EXTEND	If the VMM need use TDCALL [TDH.MR.EXTEND] for this section. 0: Do not need TDCALL [TDH.MR.EXTEND] 1: Need TDCALL [TDH.MR.EXTEND] For example, TDVF BFV sets to 1. TDVF CFV/TD_HOB/TempMem /PermMem set to 0. The Payload sets 1 or 0. The PayloadParam sets to 0.
1	PAGE.AUG	If the VMM need use TDCALL [TDH.MEM.PAGE.AUG] for this section. 0: Use TDCALL [TDH.MEM.PAGE.ADD] 1: Use TDCALL [TDH.MEM.PAGE.AUG] For example, PermMem sets be 1. Others set to 0.
31:2	Reserved	Must be 0.

Rules for the TDVF_SECTION:

- A TDVF shall include at least one BFV section and the reset vector shall be inside of BFV. The RawDataSize of BFV must be non-zero.
- A TDVF may have zero, one or multiple CFV sections. The RawDataSize of CFV must be non-zero.

- A TDVF may have zero or one TD_HOB section. The RawDataSize of TD_HOB must be zero. If TDVF reports zero TD_HOB section, then TDVF shall report all required memory in PermMem section.
- A TDVF may have zero, one or multiple TempMem sections. The RawDataSize of TempMem must be zero.
- A TD-Shim may have zero, one or multiple PermMem sections. The RawDataSize of PermMem must be zero. If a TD provides PermMem section, that means the TD will own the memory allocation. VMM shall allocate the permanent memory for this TD. TD will NOT use the system memory information in the TD HOB. Even if VMM adds system memory information in the TD HOB, it will be ignored.
- A TD-Shim may have zero or one Payload. The RawDataSize of Payload must be non-zero, if the whole image includes the Payload. Otherwise the RawDataSize must be zero.
- A TD-Shim may have zero or one PayloadParam. PayloadParam is present only if the Payload is present.

The metadata above may support below use cases as example:

- Normal TDVF: The metadata includes one BFV, one CFV, one TD_HOB and multiple TempMem.
- TD-Shim with container OS: The metadata includes one BFV, one TD_HOB, multiple TempMem and one OS kernel as Payload. The OS kernel is added so that the TD-Shim does not need load it from other storage.

TD-Shim with Service TD core: The metadata includes one BFV, zero or one CFV, multiple TempMem, one PermMem, and one Service TD Core as Payload. The TD_HOB is removed and the PermMem is added, so that the configuration is static, and all measurement registers are predictable at build time.§

12 OS Direct Boot

This section describes OS Direct Boot that may be implemented in TDVF.

The default TDVF implementation boots to an OS loader, which in turn loads the OS kernel. As implementation option, TDVF may implement an OS loader that directly boots the OS kernel.

For specialized workloads like functions or containers, the underlying guest OS is both simplified and customized. In these cases, a bootloader or general-purpose firmware is needed, so the VMM can directly boot to the guest kernel entry point. With TDX, the VMM does not choose the guest entry point and must offload the guest direct kernel boot to a TD firmware shim.

For more information on Linux Direct Boot using efi-stub, please refer to the following documentation: https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/efi-stub.rst

12.1 Measurement

The OS loader must load the OS kernel and kernel-required data (such as boot parameter) into private memory. And the OS loader must measure the kernel and required data (boot parameter) before passing the control to the OS kernel.

For example, if the TDVF loads the Linux kernel (bzImage or an ELF binary vmlinux or PVH) and initrd (initrd.img) with kernel boot parameter `console=ttyS0 root=/dev/sda4`, then TDVF needs to measure the following in RTMR[1]:

- bzImage
- initrd.img (binary format)
- `console=ttyS0 root=/dev/sda4` (string format)

Table 12-1: OS loader measurement

Data	Register	Event Log	Extended by	Checked by
OS kernel	**RTMR [1]**	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote
initrd	**RTMR [1]**	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote
Boot Parameter	**RTMR [1]**	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote
OS application	**RTMR [2]**	-	TDOS: TDCALL [TDG.MR.RTMR.EXTEND]	Remote

§

13 Minimal TDVF (TD-Shim) Requirements

This section describes requirements for a minimal TDVF implementation (TD-Shim).

In a TD container solution, the VMM loads almost everything into memory before TD launch. This means TDVF can perform minimal initialization and jump directly to the Linux Kernel without a full UEFI implementation.

For more information on Linux Boot Protocol and Linux Boot Parameter, please refer to the following documentation:

- https://www.kernel.org/doc/Documentation/x86/boot.txt
- https://www.kernel.org/doc/Documentation/x86/zero-page.txt

13.1 Hardware Virtualization-based Containers

Hardware virtualization-based containers are designed to launch and run containerized applications in hardware virtualized environments. While containers usually run directly as bare-metal applications, using TD or VT as an isolation layer from the host OS is used as a secure and efficient way of building multi-tenant Cloud-native infrastructures (e.g. Kubernetes).

In order to match the short start-up time and resource consumption overhead of bare-metal containers, runtime architectures for TD- and VT-based containers put a strong focus on minimizing boot time. They must also launch the container payload as quickly as possible. Hardware virtualization-based containers typically run on top of simplified and customized Linux kernels to minimize the overall guest boot time.

Simplified kernels typically have no UEFI dependencies and, no ACPI ASL support. This allows guests to boot without firmware dependencies. Current VT-based container runtimes rely on VMMs that are capable of directly booting into the guest kernel without loading firmware.

13.1.1 TD Container Requirements

While it is possible to let a VMM drive a VT guest via direct kernel boot, this is not part of the TD threat model (which leaves VMM outside of the TCB). The TD initial state is not modifiable by the VMM so TD containers, as opposed to VT containers, rely on a minimal TD virtual firmware solution (TD-Shim) to launch a TD guest OS.

13.2 TD-Shim Launch

The VMM launches TD-Shim in 32-bit protected mode. The TD-Shim needs to set the page tables up and then switch to 64-bit long mode.

TD-Shim assumes the VMM loads the Linux kernel (bzImage or vmLinux) and optional initrd (initrd.img) with boot parameter into memory. The VMM passes information to the TD-Shim via the TD HOB and/or Linux boot protocol and Linux boot parameter.

13.2.1 TD-Shim AP Handling

The Intel TDX module initializes all CPUs and allows them to jump to the reset vector at the same time. In a full TDVF implementation, the BSP does the TD initialization and lets APs do wait-loop. TDVF creates an ACPI table to share mailbox information with the OS, and send the OS commands via ACPI mailbox to wakeup APs.

A TD container cannot assume the guest environment supports ACPI. This enables boot to a guest environment without ACPI ASL support.

When TD containers support ACPI static table, then the TD-Shim can generate MADT table and inserts the mailbox information into it.

13.3 TD-Shim Secure Boot Support

UEFI Secure Boot is not required for TD-Shim.

13.4 TD-Shim ACPI Support

ACPI is optional for TD containers, but a typical VMM will build guest ACPI tables. When TD containers support ACPI, those tables may be passed by the initial HOB.

When TD containers do not support ACPI ASL, device information may be passed via other mechanisms (e.g. kernel command line parameter).

13.5 TD-Shim Memory Management

13.5.1 Memory Type in Initialization

Table 13-1: TD-Shim memory state from VMM

Memory Type	Provider	Data	Report	VMM Action	TDVF Action	Measurement
Static Firmware Code (Shim, Page table)	TDVF (Tenant)	Initialized	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD] SEAMCALL [TDH.MR.EXTEND]	N/A	MRTD

				SEAMCALL [TDH.MR.FINALIZE] (called after all SEAMCALL [TDH.MEM.PAGE.ADD])		
Dynamic Runtime Configuration (TD Hob)	VMM (CSP)	Initialized	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD]	TDCALL [TDG.MR.RTMR.EXTEND]	RTMR[0]
Temporary Initialized TD Memory	VMM (CSP)	0	From TD Metadata	SEAMCALL [TDH.MEM.PAGE.ADD]	N/A	N/A
Unaccepted TD Memory	VMM (CSP)	0	TD Hob - Resource Hob	SEAMCALL [TDH.MEM.PAGE.AUG]	TDCALL [TDG.MEM.PAGE.ACCEPT]	N/A
Kernel	TD Guest (Tenant)	Initialized	TD Hob - Resource Hob, GUID hob	SEAMCALL [TDH.MEM.PAGE.ADD]	TDCALL [TDG.MR.RTMR.EXTEND]	RTMR[1]
Initrd	TD Guest (Tenant)	Initialized	TD Hob - Resource Hob, GUID hob	SEAMCALL [TDH.MEM.PAGE.ADD]	TDCALL [TDG.MR.RTMR.EXTEND]	RTMR[1]
Boot Parameter	TD Guest (Tenant)	Initialized	TD Hob - Resource Hob, GUID hob	SEAMCALL [TDH.MEM.PAGE.ADD]	TDCALL [TDG.MR.RTMR.EXTEND]	RTMR[1]

13.5.2 Memory Map for OS

VMM shall pass the initial memory map information via TD HOB. TD-Shim may generate E820 table for the container.

13.6 TD-Shim Measurement

13.6.1 TD Measurement

TD-Shim must extend RTMR before transferring control to the Linux kernel.

Table 13-2: TD Measurement-Related Registers for TD-Shim

PCR	Typical Usage	Register	Event Log	Extended by	Checked by	Content
0	Firmware Code	**MRTD**	NO	VMM: SEAMCALL [TDH.MR.EXTEND]	Remote	TD-Shim + Initial Page Table

1	Firmware Data	**RTMR [0]**	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	Static Configuration (CFV), Dynamic Configuration (TD HOB)
2	Option ROM code	N/A	N/A	N/A	N/A	N/A
3	Option ROM data	N/A	N/A	N/A	N/A	N/A
4	OS loader code	**RTMR [1]**	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	OS kernel, initrd.
5	Boot Configuration	**RTMR [1]**	YES	TDVF: TDCALL [TDG.MR.RTMR.EXTEND]	Remote	Boot Parameter
6	N/A	N/A	N/A	N/A	N/A	N/A
7	Secure Boot Configuration	N/A	N/A	N/A	N/A	N/A
8~15	TD OS APP measurement	RTMR [2]	-	TDOS: TDCALL [TDG.MR.RTMR.EXTEND]	-	TD OS App. Done by OS.

13.6.2 TD Event Log

When TD containers support ACPI static table, TD-Shim passes the TD event log via ACPI table.

13.7 TD-Shim Device Support

No Device Support is required in TD-Shim. TD-Shim will jump to the Linux kernel without initializing any device.

13.8 TD-Shim Exception Handling

This model assumes TD-Shim does not generate exceptions. Exception handling is not required.

§

14 Disk Encryption

Today a VM disk image may be encrypted. Before boot, the VMM gets a storage volume key to decrypt the VM disk image and pass the decrypted disk content to the VM. However, in the TDX the VMM is not trusted. The TD needs to have a way to get the storage volume key to decrypt the VM disk image.

The VM disk image may be decrypted in OS or may be decrypted in the virtual firmware based upon the use case.

This section describes the possible solutions in the virtual firmware.

14.1 Overview

There are two agents involved in the disk encryption solution:

1) **Attestation Agent**: It will perform attestation, retrieve the storage volume key from a remote key server, and pass it to a decryption agent.

- **Preboot attestation**: The attestation agent is in TDVF or OS loader. See Figure 14-1.
- **Early OS boot attestation**: The attestation agent is in *initrd*. See Figure 14-2.
- **OS runtime attestation**: The attestation agent is an OS application.

2) **Decryption Agent**: This will get the storage volume key from the attestation agent and decrypt the disk.

The scope of the encrypted disk is use case specific. A common storage volume key may be used to decrypt:

- **Full Disk Encryption** (System Partition + Data Partition): The decryption agent may be in TDVF or integrated OS loader (grub).
- **Data Partition Encryption**: The decryption agent may be in TDVF, OS loader (grub) or initrd.
- **Container Image Encryption**: The decryption agent may be an OS application.

Table 14-1 shows these use cases.

Table 14-1: Disk Encryption Use Cases

Use Case	Preboot Attestation	Early OS Boot Attestation	OS Runtime Attestation
Full Disk Encryption	Yes	No	No
Data Partition Encryption	Yes	Yes	No
Container Image Encryption	Yes	Yes	Yes

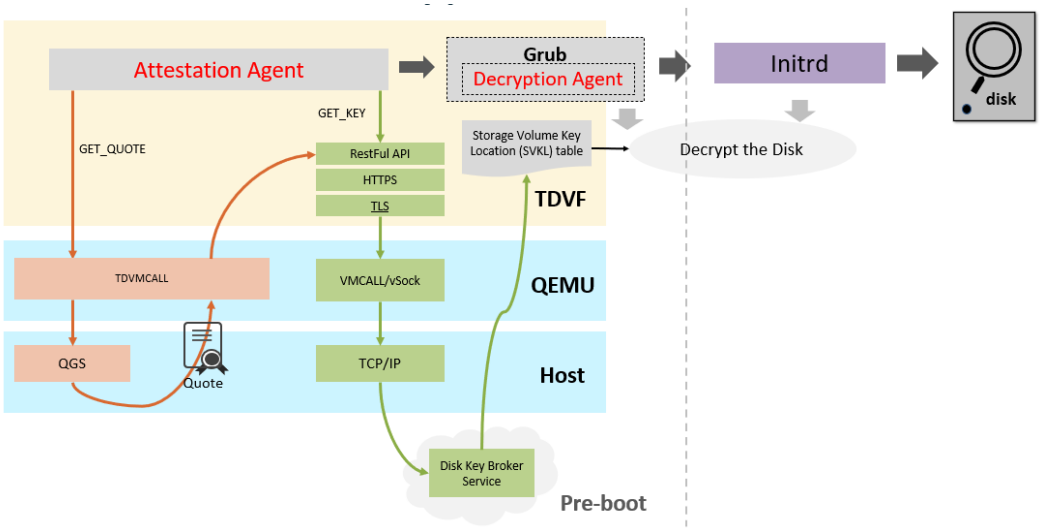

Figure 14-1: Preboot Disk Decryption Flow

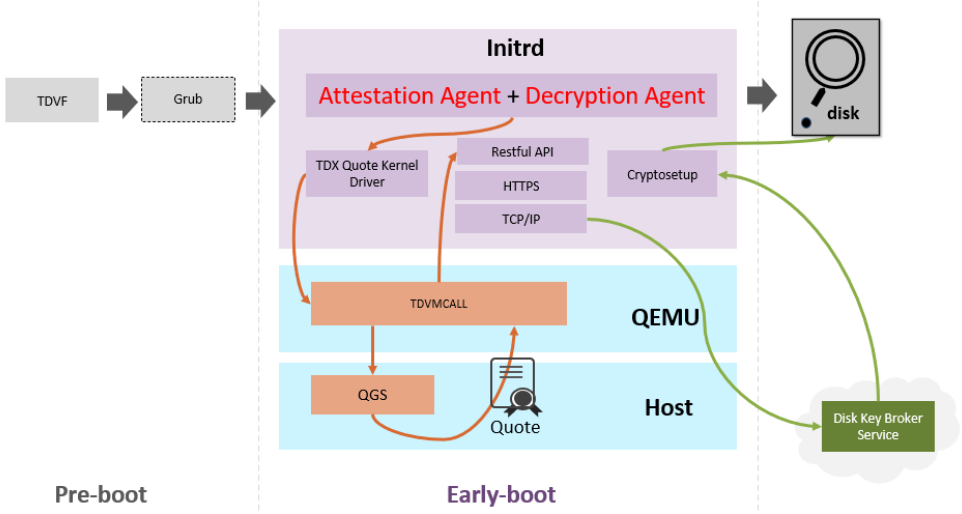

Figure 14-2: Early-boot Disk Decryption Flow

Since TDVF is involved in preboot disk decryption, we list security properties for preboot attestation as an example in table 14-2.

Table 14-2: Security Property Example

Property	**Comment**
Confidentiality	Required. VMM should not be able to read the content.
Integrity	Required. VMM should not be able to write the content.
Availability	N/A. VMM may disconnect the network for TD.
Authentication	Mutual authentication is required. 1) TD needs to ensure the data is from a real key server. 2) Key server needs to ensure the request is from a known TD.
Authorization	Simple model is used.

	Any access (read/write/execute) is allowed once the key is acquired. Key management (provision/revocation) is out of scope.
Non-Repudiation	N/A

One possible way to support the above security properties is that the TDVF attestation agent creates an authenticated secure session with the key server. Then the TDVF gets the storage volume key from the key server in the secure session.

14.2 Attestation Agent

14.2.1 Network Communication

In order to communicate with the remote key server, the TDVF needs network capability. Because the TDVF is in TCB, it is not recommended to include a full TCP/IP network stack in a TDVF. Instead, the TDVF should use the network stack in the untrusted VMM host environment.

The TDVF may use a special TDG.VP.VMCALL to send and receive the network packet on top of transport layer, such as network TLS packets, or TDVF may use the VMM specific communication, such as virt-io or vmbus.

14.2.2 Authenticated Secure Session

The network TLS protocol is a standard to allow two entities to create an authenticated secure session. TDVF can provision the public certificate of the key server and verify the server certificate at runtime.

14.2.2.1 Mutual Authentication

In order to let the key server verify the TDVF, a typical mutual authentication in TLS requires X.509 certificate provision. However, it is hard to provision a private key to TDVF. A way to resolve this problem is to use remote attestation TLS (RA-TLS).

RA-TLS does not require private key provision. The TDVF can generate an ephemeral keypair at runtime and include the public key in the TD report data and TD quote data. Then the TDVF generates an X.509 certificate at runtime, includes the TD Quote in the X.509 certificate, and sends this TD certificate to the key server as the TLS certificate. When the key server receives the certificate, it gets the TD Quote, verifies the Quote, then it can trust the public key. Finally, the key server can use the public key to verify additional TLS messages.

14.2.3 Key Server Information

The remote key server information (such as server certificate) should be provisioned to TDVF configuration FV as a UEFI variable. It will be used in TLS authentication.

This information shall be extended to RTMR for remote attestation.

14.2.4 Key transport from Server

Once TDVF and remote key server establish an authenticated secure session, the TDVF can request the key from the server. This can be done via high level network protocol on top of TLS.

One possible way is to use RESTful API. For example, the TDVF sends a "Transfer Storage Volume Key" request message, then the key server returns a "Storage Volume Key Data" response message.

14.3 Decryption Agent

14.3.1 Key Passing – SVKL table

Once the attestation agent gets the key, the attestation agent should allocate a reserved data memory and put the key into this memory, then create a Storage Volume Key Location (SVKL) ACPI table to contain the address and size of the key.

The decryption agent should locate the key from the SVKL table, read the key, then erase the key in the reserved data memory.

14.3.2 Storage Volume Key Usage in TDVF

In Full Disk Encryption, the decryption agent should decrypt the disk image. Then the TDVF can load the OS from the decrypted partition and launch it.

In Data Partition Encryption or container image encryption, the decryption happens outside of TDVF boot phase.

14.4 High-level Flow Examples

14.4.1 Example on Full Disk Encryption

1. VMM gets the encrypted disk image from the customer.

2. VMM launches TDVF with encrypted disk.

3. TDVF launches the attestation agent to communicate with remote key server.

4. The attestation agent sets up an authenticate secure session with remote key server via RA-TLS.

5. The attestation agent gets the disk encryption key from the remote key server, then passes the key to the decryption agent.

6. The decryption agent decrypts the disk.

7. TDVF locates the OS loader and boots to OS.

14.4.2 Example on Data Partition Encryption

1. VMM launches TDVF with a normal OS image and encrypted use data partition.

2. TDVF launches the attestation agent to communicate with remote key server.

3. The attestation agent sets up an authenticate secure session with remote key server via RA-TLS.

4. The attestation agent gets the storage volume key from the remote key server.

5. The attestation agent stores the storage volume key in the SVKL ACPI table.

6. TDVF boots to OS.

7. The OS decryption agent locates the SVKL ACPI table and decrypts the user data partition.

§

Appendix A - References

[TDX] Intel Trust Domain Extensions, https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[TDX-CPU] Intel CPU Architecture Extensions Specification, https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[TDX-SEAM] Intel TDX Module EAS, https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[TDX-SEAMLoader] Intel TDX Loader Interface Specification, https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[TDX-VMM] Intel TDX Guest-Hypervisor Communication Interface, https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html

[TDX Linux Security] Intel® Trust Domain Extension Linux* Guest Kernel Security Specification, https://intel.github.io/ccc-linux-guest-hardening-docs/security-spec.html**[Firmware Side Channel]** Host Firmware Speculative Execution Side Channel Mitigation, https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/host-firmware-speculative-side-channel-mitigation.html

[ACPI] ACPI Specification, https://uefi.org/specifications

[UEFI] UEFI Specification, https://uefi.org/specifications

[UEFI PI] UEFI Platform Initialization Specification, https://uefi.org/specifications

[Virtio] Virtual I/O Device (VIRTIO) Version 1.1, http://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html

[Hyper-V*] Hyper-V Top-Level Functional Specification (TLFS), https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/reference/tlfs

[TCG2 Protocol] TCG EFI Protocol specification, https://trustedcomputinggroup.org/wp-content/uploads/EFI-Protocol-Specification-rev13-160330final.pdf

[TCG ACPI] TCG ACPI specification, https://trustedcomputinggroup.org/wp-content/uploads/TCG_ACPIGeneralSpecification_v1.20_r8.pdf

[TCG PFP] PC Client-Specific Platform Firmware Profile Specification, https://trustedcomputinggroup.org/wp-content/uploads/PC-ClientSpecific_Platform_Profile_for_TPM_2p0_Systems_v51.pdf

[Linux Direct Boot]
https://github.com/torvalds/linux/blob/master/Documentation/admin-guide/efi-stub.rst

[Linux Boot Protocol] https://www.kernel.org/doc/Documentation/x86/boot.txt

[Linux Boot Parameter] https://www.kernel.org/doc/Documentation/x86/zero-page.txt

[Intel RA-TLS] Intel Remote Attestation TLS, https://github.com/cloud-security-research/sgx-ra-tls

[Open Enclave RA-TLS] open enclave Remote Attestation TLS, https://github.com/openenclave/openenclave/tree/master/samples/attested_tls, https://github.com/openenclave/openenclave/blob/master/include/openenclave/attestation/attester.h, https://github.com/openenclave/openenclave/blob/master/include/openenclave/attestation/verifier.h