Linux* Stacks for Intel® Trust Domain Extension 1.0

v0.10

July 2023

Document Number: 355388-001

Contents

Figures

TABLES

Revision History

Revision Number	Description	Date
0.8	Initial Release	1st May 2023
0.9	• Add the reference tool of check-tdx-host.sh • Add chapter 4.2.5 Linux Runtime Measurement via IMA (Linux Integrity Measurement Architecture) • Add chapter 3.5 Full Disk Encryption • Add chapter 4.4 Use Intel Project Amber • Complete the incomplete steps for Secure boot • Add chapter 6.3 Check Memory Encryption	27th May 2023
0.10	• Use $ for all command code • Normalize the "sudo" usage, use "sudo su" and simple root access used in multiple steps • Add more terminology like TCG, PCR, VMM, etc • Refine the build step for individual package build and full repo build • Update the new "Ubuntu Guest Image" tool • Add ansible deployment tool	27th June 2023

1 Introduction

1.1 Overview

Intel® Trust Domain Extension (Intel® TDX) refers to an Intel technology that extends virtual machine extensions (VMX) and Intel® Total Memory Encryption – Multi-Key (Intel® TME-MK) with a new kind of virtual machine guest called a trust domain (TD). A TD runs in a CPU mode that is designed to protect the confidentiality of its memory contents and its CPU state from any other software, including the hosting virtual machine monitor (VMM) [1]

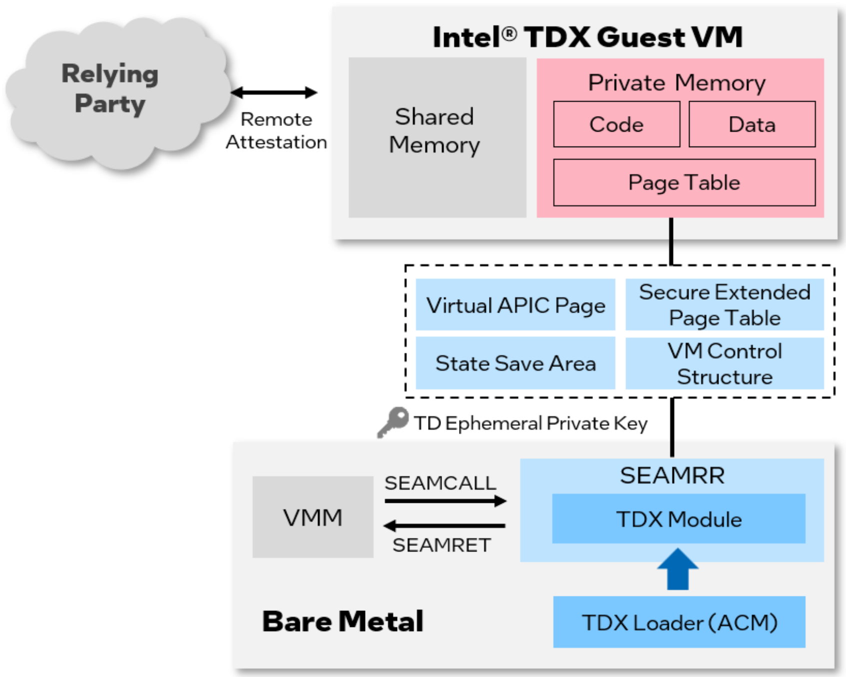
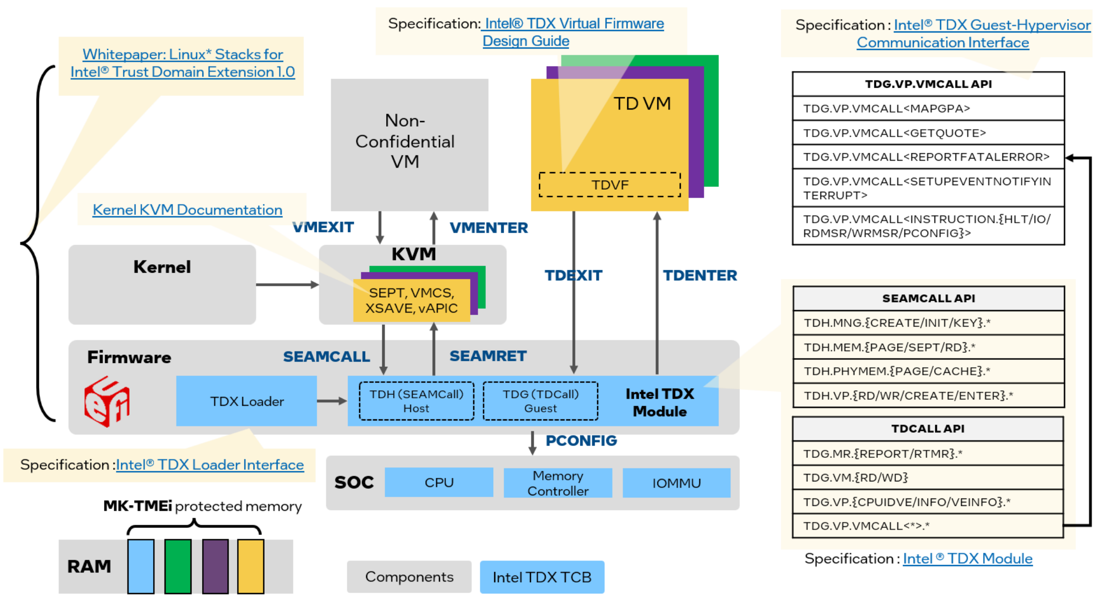


Figure 1: Intel® TDX

The white paper or specifications for Intel TDX can be found at [Intel® Trust Domain Extensions](), the major components interfaces are defined in the specifications in Figure 2: Intel TDX Component Interfaces.



Whitepaper: Linux* Stacks for Intel® Trust Domain Extension 1.0
Specification: Intel® TDX Virtual Firmware Design Guide
Specification : Intel® TDX Guest-Hypervisor Communication Interface
Non-Confidential VM
TD VM
TDVF
Kernel KVM Documentation
VMEXIT
VMENTER
KVM
SEPT, VMCS, XSAVE, vAPIC
Kernel
TDEXIT
TDENTER
Firmware
SEAMCALL
SEAMRET
TDX Loader
TDH (SEAMCall) Host
TDG (TDCall) Guest
Intel TDX Module
PCONFIG
Specification : Intel® TDX Loader Interface
SOC
CPU
Memory Controller
IOMMU
MK-TMEi protected memory
RAM
Components
Intel TDX TCB
TDG.VP.VMCALL API
TDG.VP.VMCALL<MAPGPA>
TDG.VP.VMCALL<GETQUOTE>
TDG.VP.VMCALL<REPORTFATALERROR>
TDG.VP.VMCALL<SETUPEVENTNOTIFYINTERRUPT>
TDG.VP.VMCALL<INSTRUCTION.{HLT/IO/RDMSR/WRMSR/PCONFIG}>
SEAMCALL API
TDH.MNG.{CREATE/INIT/KEY}.*
TDH.MEM.{PAGE/SEPT/RD}.*
TDH.PHYMEM.{PAGE/CACHE}.*
TDH.VP.{RD/WR/CREATE/ENTER}.*
TDCALL API
TDG.MR.{REPORT/RTMR}.*
TDG.VM.{RD/WD}
TDG.VP.{CPUIDVE/INFO/VEINFO}.*
TDG.VP.VMCALL<*>.*
Specification : Intel ® TDX Module

Figure 2: Intel TDX Component Interfaces

Linux* Stacks for Intel® TDX is an end-to-end hypervisor cloud stack including the Infrastructure as a Service (IaaS) and Platform as a Service (PaaS) components to produce the following minimal use cases:

- Launch Intel® TDX guest VM to run general computing workloads.
- Do launch-time measurement within the Intel® TDX guest VM.
- Do runtime attestation with the quote generated by Intel® Software Guard Extensions (Intel® SGX)-based quote generation service (QGS) on the IaaS host.

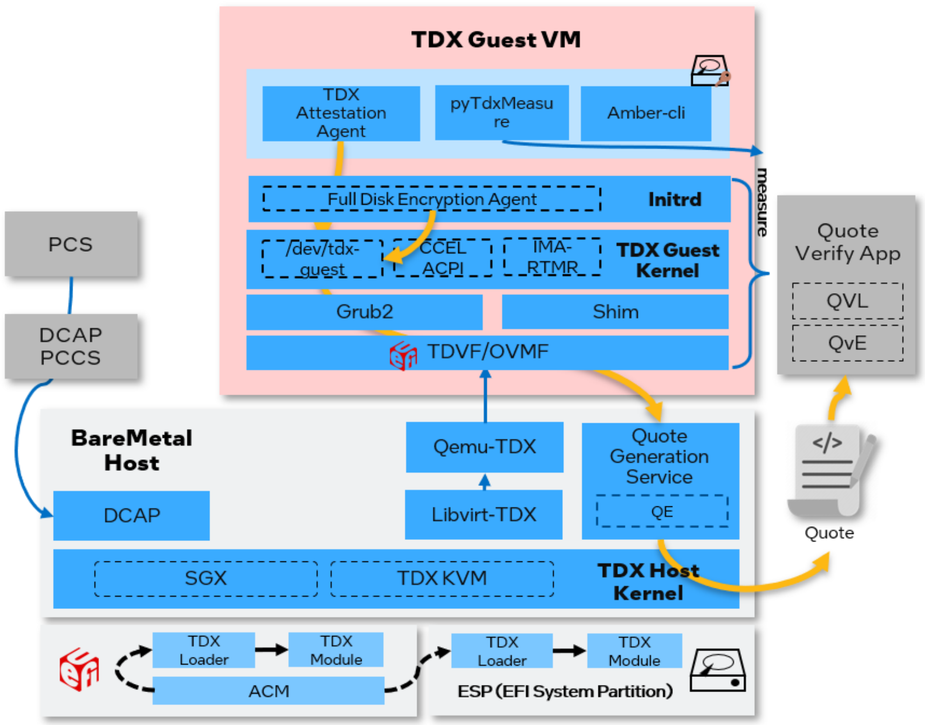


Figure 3: Linux Stack for Intel TDX

The open-source code for Linux Stack for Intel TDX can be found at:

https://github.com/intel/tdx-tools.

NOTE: tdx-tools has multiple release tags*. Please make sure to use the correct release tag which matches the release version. The Release tag and keron mapping can be th* tdx-tools wiki.

This document introduces:

- The deployment, cloud stack test, and other common uses for those who want to validate confidential workloads or tune performance.
- The debug and development methods for those who want to integrate stack in their IaaS/PaaS framework.

1.2 Terminology

Term	Description
ACM	Authenticated Code Module
CFV	Configuration Firmware Volume
CMR	Convertible Memory Ranges
CPLD	Complex Programmable Logic Device
CRB	Customer Reference Board
DCAP	Data Center Attestation Primitives
DIMM	Dual In-line Memory Module
ECC	Error Correction Code memory
ESP	EFI System Partition
GPA	Guest Physical Address
GVA	Guest Virtual Address
HVC	Hypervisor Virtual Console
IFWI	Integrated Firmware Image
IBV	Independent BIOS Vendor
Intel SGX	Intel® Software Guard Extensions (Intel® SGX)
Intel TDX	Intel® Trust Domain Extension (Intel® TDX)
LIV server	Live server is used for attestation with production CPU SKUs
LUKS	Linux Unified Key Setup
MOK	Machine Owner Key
MRTD	Measurement of Trust Domain Firmware
OVMF	Open-source Virtual Machine Firmware
PCR	Platform Configuration Register
PCS	Provisioning Certification Service
PCCS	Provisioning Certificate Caching Service
PK/KEK/DB	Platform Key/Key Exchange Key/Database Key
QMP	QEMU Monitor Protocol
RTMR	Runtime Measurement Register
SBX server	Sandbox server is used for attestation with pre-production CPU SKUs
SEAM	Secure Arbitration Mode
SVN	Security Version Number
TCB	Trusted-Computing Base
TCG	Trusted Computing Group
TDVF	Trusted Domain Virtual Firmware
TDVM	A TD guest VM

TEE	Trusted Execution Environment
VMM	Virtual Machine Monitor

2 Install

2.1 Hardware

Linux Stacks for Intel TDX needs the following hardware support that enables Intel TDX:

- CPU Processor SKU. Contact Intel sales representative for details.
- Board configurations via hardware jumper or CPLD (complex programmable logic device). Contact your ODM/OEM vendor.
- DDR5 DIMM with the type of 10 × 4 ECC (error correction code memory)
- DDR5 RDIMMs with integrity protection.
- DIMM (Dual in-line memory module) population: It is recommended that all channel 0 slots have at least 1 DIMM populated. There are a total of 8 DIMMs per socket. DIMM population must be symmetric across the integrated memory controllers.

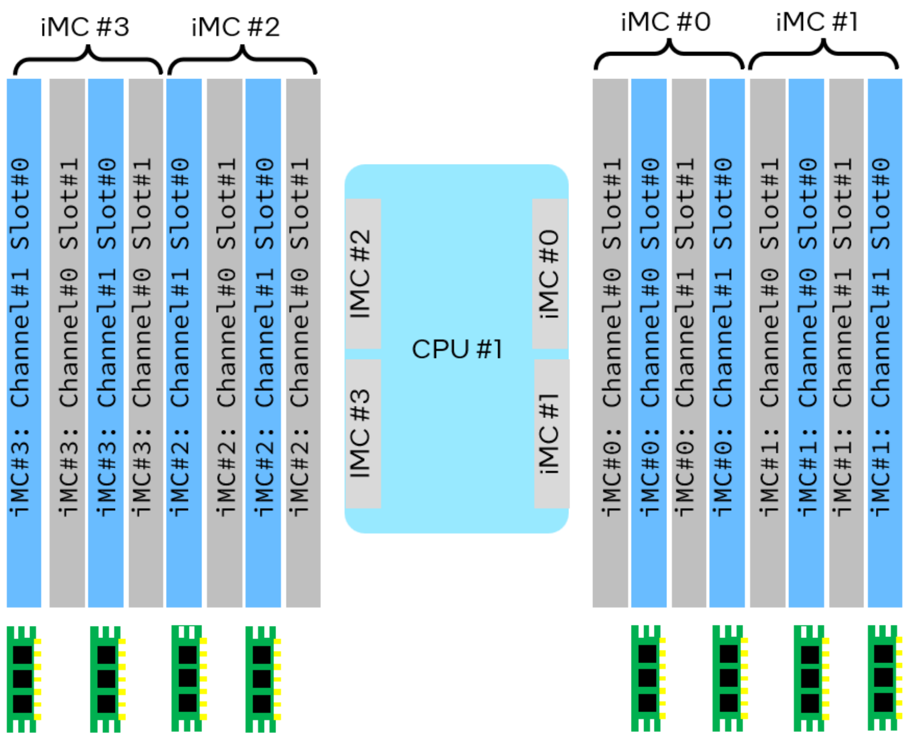


Figure 4: 8+0 DIMM Population for Intel TDX

Intel TDX also supports the full DIMM population 16+0 as follows:

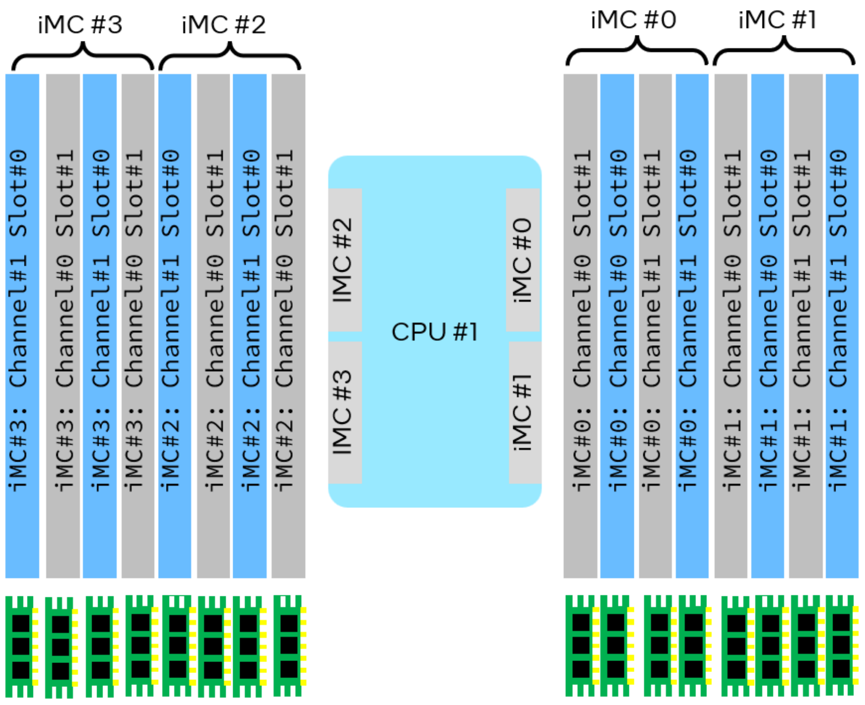
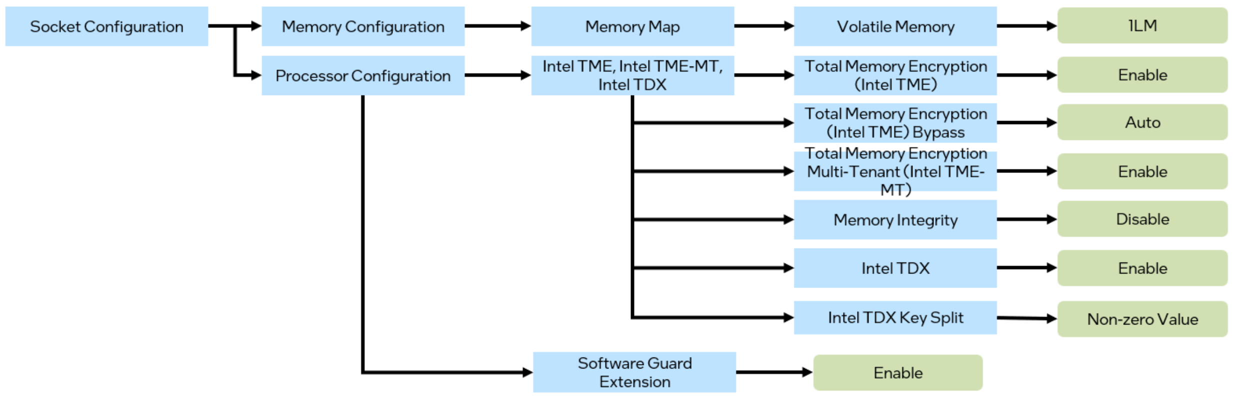


Figure 5: 16+0 DIMM Population for Intel TDX

2.2 BIOS

BIOS configurations are needed to support Intel TDX. Contact an Intel sales representative or IBV (independent BIOS vendor) for details. The following settings are examples for reference:



Socket Configuration → Memory Configuration → Memory Map → Volatile Memory → 1LM

Socket Configuration → Processor Configuration → Intel TME, Intel TME-MT, Intel TDX → Total Memory Encryption (Intel TME) → Enable

Socket Configuration → Processor Configuration → Intel TME, Intel TME-MT, Intel TDX → Total Memory Encryption (Intel TME) Bypass → Auto

Socket Configuration → Processor Configuration → Intel TME, Intel TME-MT, Intel TDX → Total Memory Encryption Multi-Tenant (Intel TME-MT) → Enable

Socket Configuration → Processor Configuration → Intel TME, Intel TME-MT, Intel TDX → Memory Integrity → Disable

Socket Configuration → Processor Configuration → Intel TME, Intel TME-MT, Intel TDX → Intel TDX → Enable

Socket Configuration → Processor Configuration → Intel TME, Intel TME-MT, Intel TDX → Intel TDX Key Split → Non-zero Value

Socket Configuration → Processor Configuration → Software Guard Extension → Enable

Figure 6: BIOS settings for Intel TDX

Table 1: Intel TDX BIOS Configurations

BIOS Setting	Notes
Volatile Memory = 1LM	Intel TDX and CMR (Convertible Memory Ranges) logical integrity, isolation, and cryptographic integrity are only available with directly attached DDR5 memory.
Total Memory Encryption (Intel TME) = Enable	Intel TDX technology depends on Intel® Total Memory Encryption (Intel® TME).
Total Memory Encryption (Intel TME) Bypass = Auto	4th generation Intel Xeon Scalable processors introduce an Intel TME bypass mode to allow memory outside of Intel TME multi-tenant virtual machines, Intel SGX enclaves, and Intel TDX trust domains to be unencrypted to improve the performance of nonconfidential software.
Total Memory Encryption Multi-Tenant (TME-MT) = Enable	128 Intel TME – Multi-Tenant encryption keys.
Memory Integrity = *Disable*	4th generation Intel Xeon Scalable processor E-stepping does not support Intel TDX-CI, but only supports Intel TDX-LI.
Intel TDX = Enable	Intel TDX should be enabled.
TDX Key Split = <Non-zero Value)	Keys split between Intel TME multi-tenant and Intel TDX.
Software Guard Extension = *Enable*	Intel TDX depends on Intel SGX technology for hardware TCB and remote attestation.

Note: The configuration or the menus might be different on your BIOS. Contact the IBV or OEM/ODM for the correct settings.

2.3 Components

Linux Stack for Intel TDX is a vertical end-to-end stack including a series of components which are listed in Table 2: Linux Stack for Intel TDX Components Table 2: Linux Stack for Intel TDX Components.

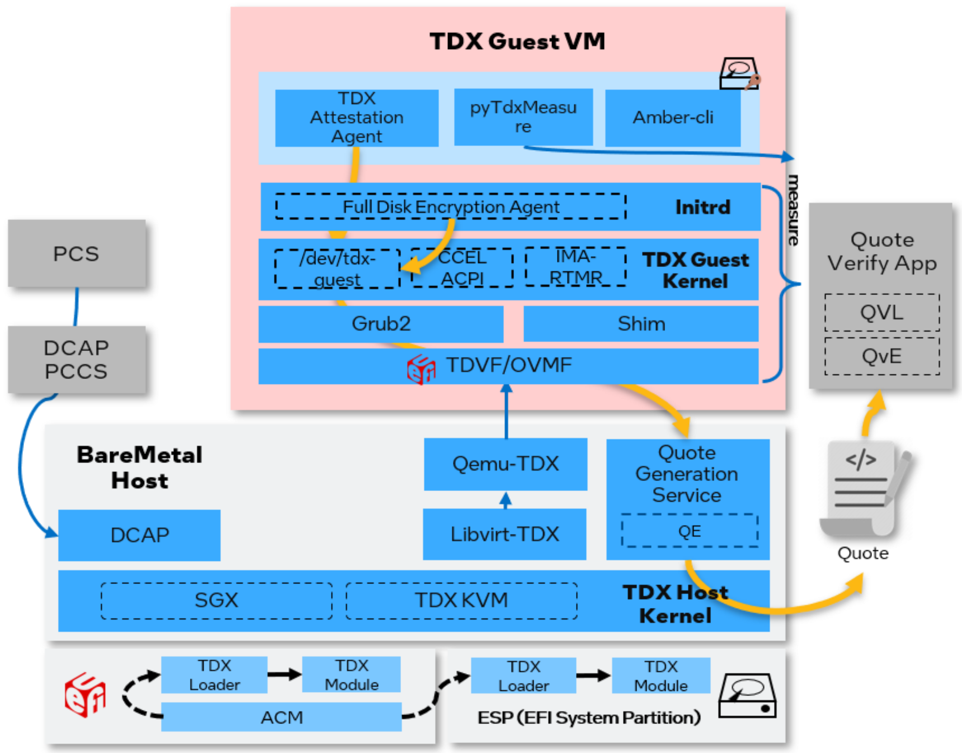


Figure 7: Linux Stack for Intel TDX overview

There are multiple components in the Linux stack for both the host and guest.

Table 2: Linux Stack for Intel TDX Components

Components	Description	Source
Intel TDX Module	An attested software module running in SEAM Root Mode.	Intel Trust Domain Extensions
TDX Loader	A SEAM module intended to install an Intel TDX module into SEAM range.	Intel Trust Domain Extensions
Intel TDX Host Kernel	The host kernel with Intel TDX patches being upstreamed.	https://github.com/intel/tdx/tree/kvm
Intel TDX QEMU	QEMU with Intel TDX patches being upstreamed.	https://github.com/intel/QEMU-tdx
Intel TDX Libvirt	Libvirt with Intel TDX patches being upstreamed.	https://github.com/intel/libvirt-tdx/tree/for_QEMU_upstream
TDVF	Virtual firmware (aka OVMF) with Intel TDX features already upstreamed.	https://github.com/tianocore/edk2
DCAP	Intel SGX-based DCAP (data center attestation primitives)	https://github.com/intel/SGXDataCenterAttestationPrimitives

	for the platform certificate after registration.	
QGS	QGS provides the functionality of Intel TDX quote generation within an Intel SGX-based quote enclave. It is part of the DCAP running on the IaaS host or legacy VM.	https://github.com/intel/SGXDataCenterAttestationPrimitives
Intel TDX Guest Kernel	The guest kernel with Intel TDX patches being upstreamed.	https://github.com/intel/tdx/tree/guest-upstream
Grub2	The bootloader grub2 with Intel TDX patches already upstreamed.	https://github.com/intel/grub-tdx/tree/2.06-upstream-v4
Shim	The bootloader shim with Intel TDX patches already upstreamed.	https://github.com/intel/shim-tdx
Intel TDX Attestation Agent	A sample Intel TDX attestation agent to call TDVMCALL.getQuote(). It is part of DCAP.	https://github.com/intel/SGXDataCenterAttestationPrimitives
PyTdxMeasure	A Python measurement library that dumps RTMR, the CCEL ACPI table, and verifies the RTMR via replaying the TD event log.	https://github.com/intel/tdx-tools

NOTE: Some of the components have completed patch upstreaming such as Grub, Shim, and TDVF, while others are still in progress.

2.4 Building Stacks

For an end-to-end stack setup and validation, tdx-tools provides downstream patches and a build tool to construct the whole stack in a few simple steps.

The supported distros' versions are as follows for both host and guest packages:

- RHEL 8.x (Latest Version)
- Ubuntu 22.04

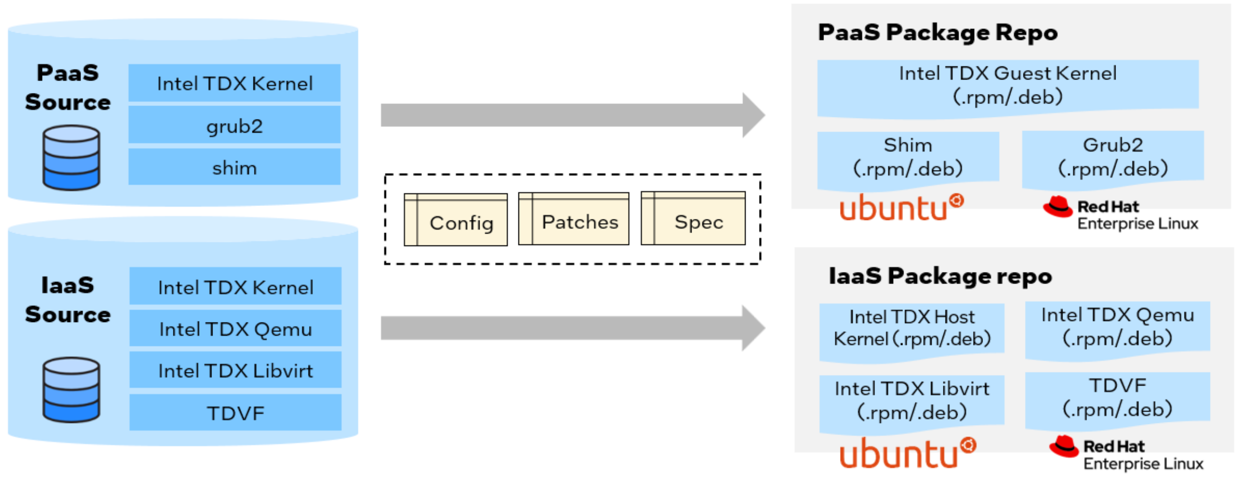


Figure 8: End-to-End Host and Guest Stack for Linux and Intel TDX

The end-to-end stack building includes two steps on any Linux development machine:

- Step 1: Build packages
- Step 2: Create guest image

2.4.1 Build Packages

A `build.sh` script is provided by tdx-tools to download the upstream source, apply Intel TDX patches from the directory `<build>/common`, and do package building.

Note: When obtaining tdx-tools, please make sure to use the correct tag which matches the release version.

Below is an example of how to build individual packages on Ubuntu 22.04.

```
$ cd build/ubuntu-22.04/
$
$ # The following will only build the intel-mvp-tdx-kernel package
$ ./pkg-builder intel-mvp-tdx-kernel/build.sh
```

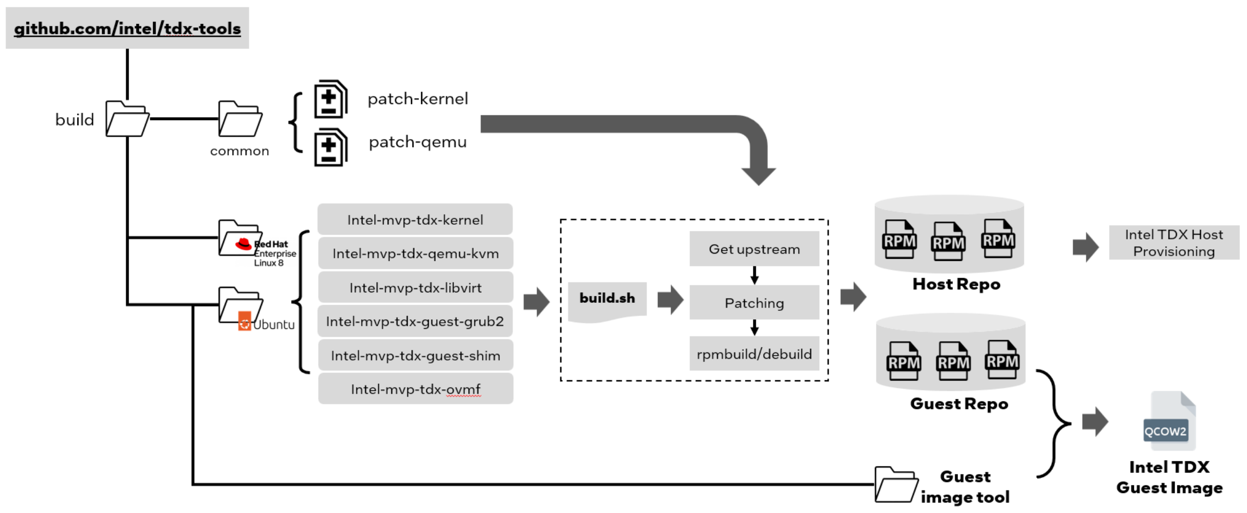


Figure 9: Build process for Intel TDX packages

The kernel config is provided in the kernel package directory with Intel TDX configurations. For example, RHEL-8 will use the path `build/rhel-8/intel-mvp-tdx-kernel/tdx-kernel.spec`. All kernel configurations have been optimized for TDX performance.

Below is an example using Ubuntu 22.04 to build the entire repo with all the packages.

```
$ cd build/ubuntu-22.04/
$
$ # Build the all packages via build-repo.sh, and running it within pkg-builder
$ # container to avoid any issues of build environment
$ ./pkg-builder build-repo.sh
$
$ # If want to build individual package like intel-mvp-tdx-kernel
$ ./pkg-builder intel-mvp-tdx-kernel/build.sh
$
```

After the packages have been built successfully, two repositories are generated.

1. `build/ubuntu-22.04/host_repo`: includes the Intel TDX host kernel, Intel TDX QEMU, Intel TDX Libvirt, and TDVF.
2. `build/ubuntu-22.04/guest_repo` : includes the Intel TDX guest kernel, grub2, and shim.

2.4.2 Create Guest Image

The Intel TDX virtual machine requires guest images to be booted with a Intel TDX guest kernel, Intel TDX grub2 and shim packages for both grub boot and secure boot.

The image creation process is a little different for the different distros:

- RHEL 8.x
  The default distro cloud image does not support an EFI schema. The tdx-tools repository provides `<tdx-tools>/build/rhel-8/guest-image/create-efi-img.sh` to create a RHEL EFI guest image via the kickstart[1] tool.

  Use `tdx-guest-stack.sh` to install the Intel TDX guest packages into the image using the following workflow.

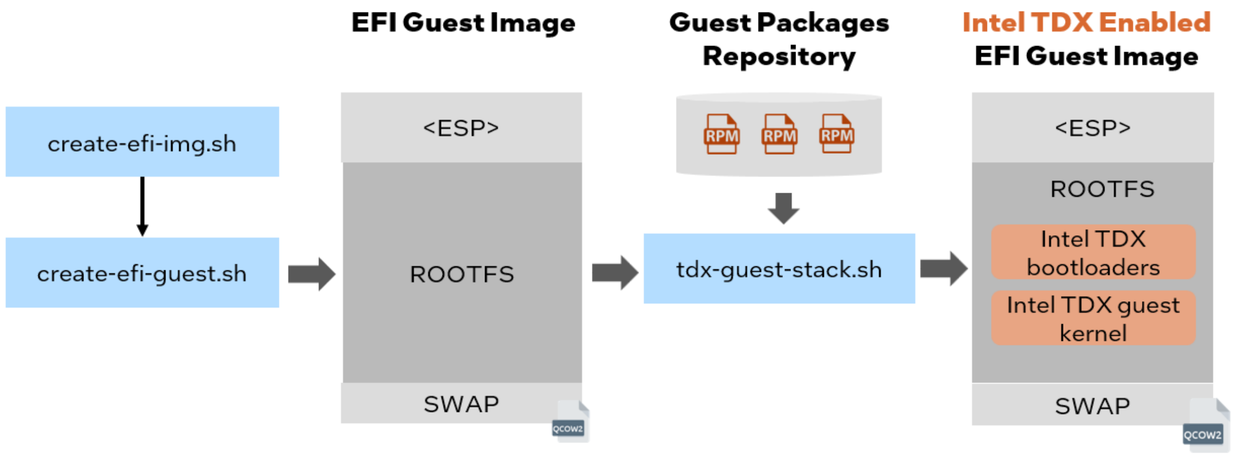


Figure 10: Create Intel TDX RHEL guest image

- Ubuntu 22.04

  Ubuntu provides EFI enabled guest/cloud images at https://cloud-images.ubuntu.com/. Use `<tdx-tools>/build/ubuntu-22.04/guest-image/create-ubuntu-image.sh` as follows:

```
$ # Prerequisite: build the Ubuntu packages via build-repo.sh
$
$ cd build/ubuntu-22.04/guest-image
$
$ # Install additional packages into guest image
$ ./create-ubuntu-image.sh
```

[1] https://linuxhint.com/beginners-kickstart/

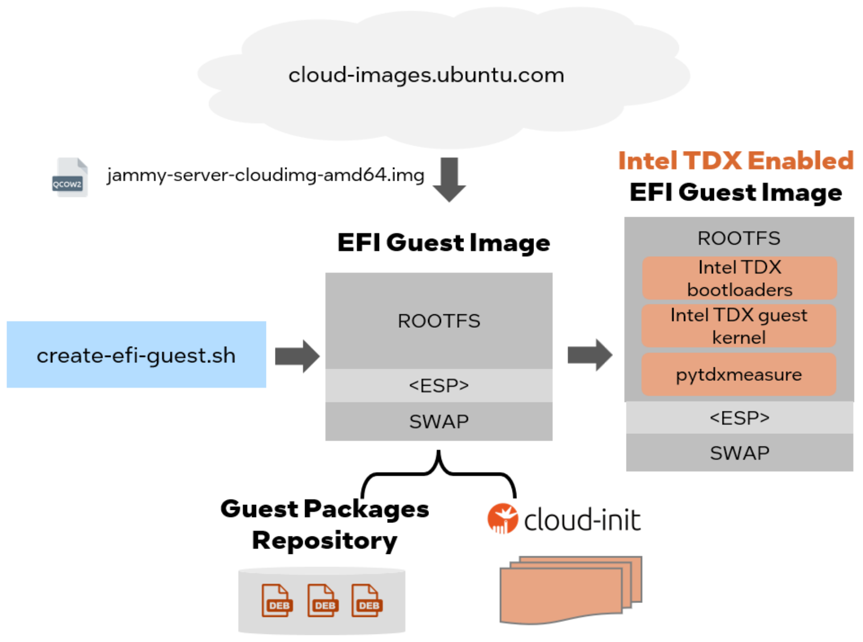


Figure 11: Create Intel TDX Ubuntu Guest Image

NOTE: The guest image can be further customized at tdx-tools/build/ubuntu22.04/guest-image:

```
Usage: create-ubuntu-image.sh [OPTION]...
  -h                            Show this help
  -c                            Create customize image (not from Ubuntu official cloud
image)
  -f                            Force to recreate the output image
  -n                            Guest host name, default is "tdx-guest"
  -u                            Guest user name, default is "tdx"
  -p                            Guest password, default is "123456"
  -s                            Specify the size of guest image
  -o <output file>              Specify the output file, default is tdx-guest-ubuntu-
22.04.qcow2.
                                Please make sure the suffix is qcow2. Due to
permission consideration,
                                the output file will be put into /tmp/<output file>.
  -r <guest repo>               Specify the directory including guest packages,
generated by build-repo.sh
```

2.5 Install IaaS Host

Perform the following steps to deploy the packages on IaaS host.

NOTE: Please disable Intel TDX in the BIOS to install the Intel TDX packages. Before installing the Intel TDX host kernel, the distro default kernel may unintentionally cleanup MTRR (memory type range register) for the Intel TDX memory range. This will cause an MCHECK error. Once the Intel TDX packages *have been installed, set the Intel TDX kernel as the default in grub, reboot, and enable Intel TDX in the BIOS.*

2.5.1 Install Packages by Manual

For RHEL 8.x host

- Move the generated host repo to a directory that will be used in the repo file.

```
$ sudo mkdir -p /srv/
$
$ # Build the RPM packages via above steps, the RPM package will be generated
$ # in <tdx-tools>/build/rhel-8/host_repo/
$ sudo mv build/rhel-8/host_repo/ /srv/tdx-host
```

- Set up the host repository. Generate the file /etc/yum.repos.d/tdx-host-local.repo and add the following content.

```
$ cat /etc/yum.repos.d/tdx-host-local.repo
[tdx-host-local]
name=tdx-host-local
baseurl=file:///srv/tdx-host
enabled=1
gpgcheck=0
module_hotfixes=true
```

- Add the EPEL repo. It provides the packages of `capstone` and `libcapstone` required by Intel TDX QEMU.

```
$ sudo dnf install https://dl.fedoraproject.org/pub/epel/epel-releaselatest-
8.noarch.rpm
```

- Install the host packages.

```
$ sudo dnf install intel-mvp-tdx-kernel intel-mvp-tdx-QEMU-kvm intel-mvp-ovmf
intel-mvp-tdx-libvirt
```

- If you get an error about QEMU-kvm conflicts, remove the existing QEMU-kvm package with the following command, and then re-run the command above to install host packages.

```
$ sudo dnf remove QEMU-kvm
```

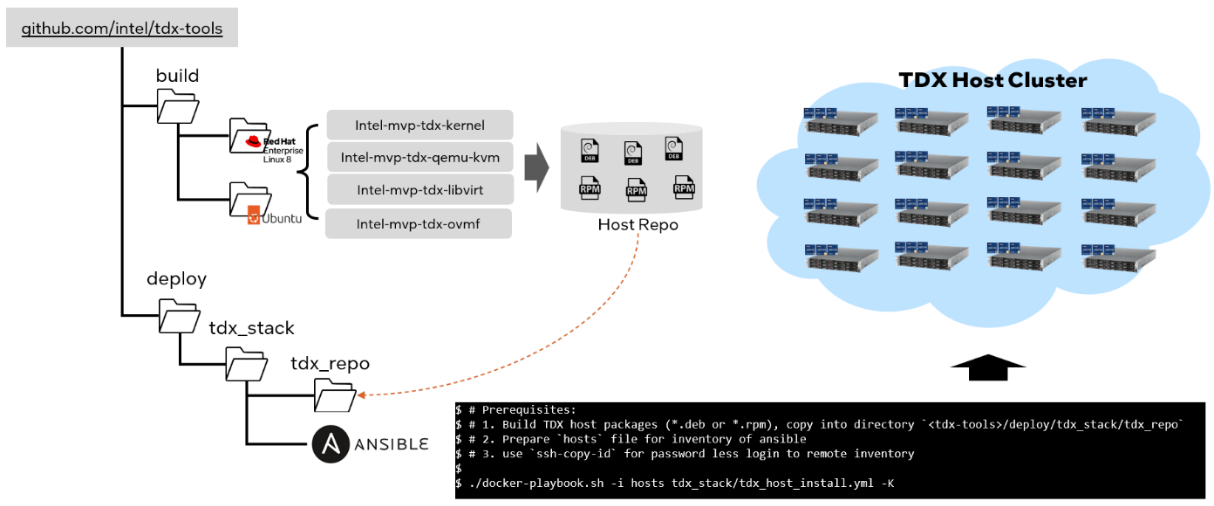
For Ubuntu 22.04 host

- Install all Debian packages

```
$ # Build the DEB packages via the above steps, the DEB packages will be generated
$ # to <tdx-tools>/build/ubuntu-22.04/host_repo/
$ cd host_repo
$ sudo apt -y --allow-downgrades install ./*.deb
```

2.5.2 Deploy via Ansible

Use the ansible based "TDX Deployment Tool" to deploy the host stack to multiple server nodes or guest stack to multiple TD VMs.



```
$ # Prerequisites:
$ # 1. Build TDX host packages (*.deb or *.rpm), copy into directory `<tdx-tools>/deploy/tdx_stack/tdx_repo`
$ # 2. Prepare `hosts` file for inventory of ansible
$ # 3. use `ssh-copy-id` for password less login to remote inventory
$
$ ./docker-playbook.sh -i hosts tdx_stack/tdx_host_install.yml -K
```

Figure 12: Deploy TDX host stack via Ansible

NOTE: Current Ansible script only support deployment on Ubuntu22.04.

Before using the script there are a couple pre-requisites.

1. Build the host *.deb repositories. Refer to section 2.4.1 Build Packages. The final built files will be at `<tdx-tools>/build/ubuntu-22.04/host_repo`.
2. Copy all *.deb packages to `<tdx-tools>/deploy/tdx_stack/tdx_repo/`

```
$ cp build/ubuntu-22.04/host_repo/*.deb deploy/tdx_stack/tdx_repo/
```

3. Prepare ansible inventory host file and enable password-less login on all managed nodes, please refer tutorial.

```
$ # The content of an example inventory file
$ cat deploy/tdx_stack/hosts
tdx@10.0.0.2
tdx@10.0.0.3
tdx@10.0.0.4

$ # Enable the SSH password less login
$
$ ssh-copy-id tdx@10.0.0.2
...
$ ssh-copy-id tdx@10.0.0.3
...
$ ssh-copy-id tdx@10.0.0.4
```

4. Setup ansible docker to create the build image

```
$ ./docker-playbook.sh rebuild
```

5. Run playbook in docker for auto deployment process on all managed nodes

```
$ ./docker-playbook.sh -i hosts tdx_stack/tdx_host_install.yml -K
```

2.5.3 Configure Grub

Add `numa_balancing=disable` into grub menu.

For RHEL 8.x

```
$ vi /etc/default/grub

# Add "numa_balancing=disable" in GRUB_CMDLINE_LINUX
GRUB_CMDLINE_LINUX=". . . numa_balancing=disable"

$ sudo grub2-mkconfig -o /boot/efi/EFI/redhat/grub.cfg
```

For Ubuntu 22.04

```
$ vi /etc/default/grub

# Add "numa_balancing=disable" in GRUB_CMDLINE_LINUX_DEFAULT
GRUB_CMDLINE_LINUX_DEFAULT=". . . numa_balancing=disable"

$ sudo update-grub
```

2.5.4 Set Default Kernel

For RHEL 8.x

```
$ ls /boot/vmlinuz* | grep <kernel-version>

$ # Use output of above command, such as
$ # "/boot/vmlinuz-6.2.0-tdx.v1.5.mvp7.el8.x86_64" in the
$ # below command

$ sudo grubby --set-default=/boot/vmlinuz-<kernel version>
```

For Ubuntu 22.04

```
$ grep -A100 submenu /boot/grub/grub.cfg | grep menuentry | grep <TDX kernel
version>

$ # Use the string in above output, such as "gnulinux-6.2.0-mvp4v1+2-
$ # generic-advanced-34db9317-bf73-44c3-8425-2fa83446e8d5" in
$ # /etc/default/grub file as value of “GRUB_DEFAULT"

$ vi /etc/default/grub
GRUB_DEFAULT="gnulinux-6.2.0-mvp4v1+2-generic-advanced-34db9317-bf73-44c3-8425-
2fa83446e8d5"
```

```
$ sudo update-grub
```

NOTE: Change the default string according to your TDX kernel version.

2.5.5 Reboot with the Intel TDX kernel

After installing the Intel TDX kernel and host packages successfully, reboot the system into the BIOS menu and enable Intel TDX. Refer to Section 2.2 on the BIOS settings. To verify that Intel TDX is enabled by using the script check-tdx-host.sh after the system is booted. Alternatively, the below steps can also be used to determine that Intel TDX has been successfully enabled.

- Check whether TDX Module is initialized. The expected output is "TDX module initialized".

```
$ sudo dmesg | grep -i tdx
...
tdx: TDX module initialized.
```

- Check Intel TME enable status; expect a return code of 1.

```
$ sudo rdmsr -f 1:1 0x982
1
```

- Check Intel TME max keys.

```
$ sudo rdmsr -f 50:36 0x981
```

- Check the Intel SGX and MCHECK status, expecting a code of 0.

```
$ sudo rdmsr 0xa0
0
```

- Check the Intel TDX Status, expecting a code of 1.

```
$ sudo rdmsr -f 11:11 0x1401
1
```

- Check the number of Intel TDX keys

```
$ sudo rdmsr -f 63:32 0x87
1
```

- Check the information for the Intel TDX module.

```
$ cat /sys/firmware/tdx/tdx_module/*
```

3 Manage the TD guest

Like a normal virtual machine, a TD guest can be launched by QEMU via command line or orchestrated by Libvirt via XML templates. This chapter introduces how to manage the lifecycle of a TD guest for diverse boots such as secure boot, direct boot, and grub boot.

NOTE: Please make sure to use the correct tag of tdx-tools which matches the release version.

3.1 Overview

You can boot a TD guest either by using the QEMU command line or by using a Libvirt XML template and virsh commands. Libvirt translates the XML template to QEMU commands and calls QEMU-kvm to complete the VM boot. Similarly, you can call QEMU-kvm directly with parameters to boot a VM.

The following diagram illustrates the TD guest boot type and boot process.

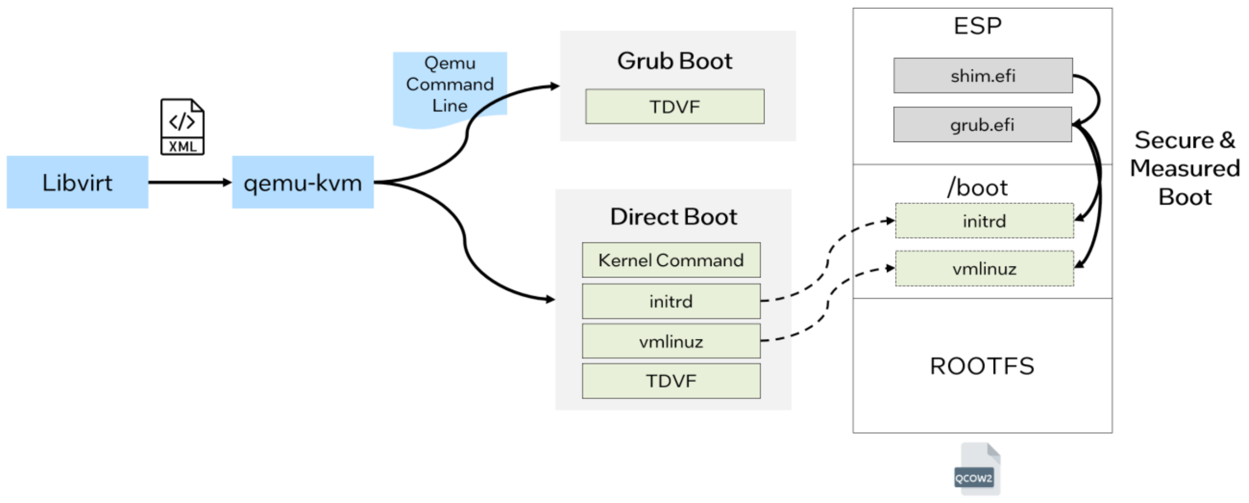


Figure 13: TD Guest Boot Process

The following table explains different boot types:

Table 3: Boot Types for TD Guest

Boot Type	Description	Difference from a non-confidential VM
Direct Boot	Boot the guest by explicitly specifying kernel binary via QEMU launching parameter "-kernel", specifying the initrd binary via QEMU launching parameter "-initrd", specifying the kernel command via QEMU launching parameter "-append". Bootloaders, such as shim/grub are not involved in direct boot.	No differences on QEMU launch parameters for confidential VM, but requires TDVF/OVMF to do measured boot and record the measurement into RTMR(Runtime Measurement Register) register
Grub Boot	Boot the guest without "-kernel" and "-append" in QEMU launching params. The OVMF/TDVF searches for and starts the bootloader from ESP	No differences on QEMU launch parameters for confidential VM, but requires Intel TDX Grub2 to do measured boot and record the measurement into RTMR register
Measured Boot	It is the process of measuring and storing securely (i.e. using a TPM) the next stage object in the boot process by the UEFI BIOS, bootloader, kernel, etc.	The secure register is a PCR register in a trusted computing group (TCG)-defined trusted platform module (TPM), while is RTMR in Intel TDX module
Secure Boot	Secure boot is a security standard developed by members of the PC industry to help make sure that a device boot using only software that is trusted by the original equipment manufacturer (OEM). The Secure boot certificate should be protected by measured boot.	The secure boot certificate can be enrolled in runtime of guest VM for non-confidential VM, while the secure boot must be enrolled in TDVF offline for the consistent measurement on MRTD (measurement of trust domain)

The detail boot flow for different TD boot methods can be found in Figure 14: Detailed boot flow for different TD boot

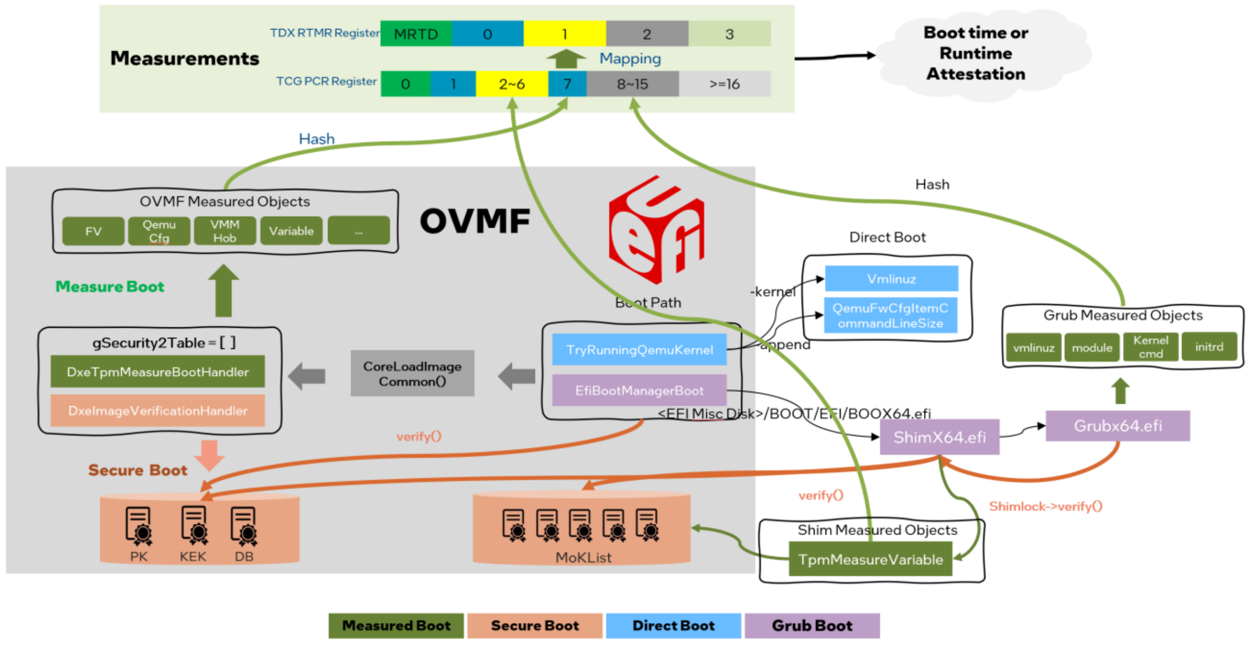


Figure 14: Detailed boot flow for different TD boot

In Figure 14: Detailed boot flow for different TD boot:

Measured Boot

1. In OVMF, the image handler `DxeTpmMeasureBootHandler` will be triggered when loading EFI image via `CoreLoadImageCommon()`.
2. In OVMF, `DxeTpmMeasureBootHandler` will measure the objects like FV, QEMU CFG, VMM Hob, Variable into TCG PCR Register. In TD guest, the measurement will also be extended to RTMR register if vTPM does not exist.
3. In boot loader ShimX64.efi, `TpmMeasureVariable()` will measure the secure boot's certificates into TCG PCR or TDX RTMR register.
4. Boot loader GrubX64.efi measures kernel binary and cmdline, initrd binary and grub's module into TCG PCR or TDX RTMR register.

The mapping between TCG PCR register and RTMR is

➢ PCR #1, #7 ⇔ RTMR #0
➢ PCR #2~#6 ⇔ RTMR #1
➢ PCR #8~#15 ⇔ RTMR #2

Secure Boot

1. In OVMF, the image handler `DxeImageVerificationHandler` will be triggered when loading EFI image via `CoreLoadImageCommon()`

2. In OVMF, `DxeImageVerificationHandler` will use UEFI secure boot's DB key verify the certificate from each of load EFI image.
3. ShimX64.efi and GrubX64.efi will use UEFI secure boot's DB key or Linux secure boot's MoK (Machine Owner Key) to verify the certificate of kernel, kernel module etc.
4. All certificates, including UEFI secure boot and Linux secure boot, are measured into TDX RTMR register.

Direct Boot

1. In direct boot path, the kernel binary is started and measured by `TryRunningQEMUKernel()`.
2. In direct boot path, the kernel command and initrd are measured by the EFI stub of Linux kernel

Refer to the measure log for direct boot for a sample of what the output should look like.

Grub Boot

1. In grub boot, ShimX64.efi will help bring the UEFI secure boot to Linux secure boot. Normally UEFI secure boot will use MSFT UEFI certificates[2]. But here it needs a new key to sign customized kernel/grub/shim, please refer 3.3 Secure Boot
2. In grub boot, GrubX64.efi will measure kernel binary/command and initrd binary.

Refer the measure log for grub boot.

[2] https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance?view=windows-11

3.2 Boot TD Guest

3.2.1 Launch via QEMU

Since the QEMU parameter list is quite long and complicated, tdx-tools provides the start-qemu.sh script to handle some parameters by default. It supports both direct boot and grub boot of TD guest. It also provides a few interactive parameters to meet customization requirements.

Note: The parameters in "start-qemu.sh" may vary along with different Intel TDX kernel and Intel TDX QEMU versions. Please make sure to use the correct tag of tdx-tools which matches the release.

The start-qemu.sh script offers several parameters to boot TD guests on demand. The parameters are listed in Table 4: start-qemu.sh parameters:

Table 4: start-qemu.sh parameters

Parameter	Description
`-i <guest image file>`	Guest image file name and location
`-k <kernel file>`	Kernel binary name and location
`-t [legacy | efi | td]`	VM type supported; default is "td"
`-b [direct | grub]`	Boot type, default value is "direct" which requires kernel binary specified via "-k"
`-p <Monitor port>`	Monitor port via telnet. Refer to the usage of QEMU Monitor
`-f <SSH Forward port>`	Host port used for guest VM SSH forwarding. Refer to QEMU SSH port forwarding
`-o <OVMF file>`	BIOS firmware device file. OVMF/TDVF is used for "td" and "efi" VM type. "efi" is used for non-confidential VM, while "td" is used for TD VM guest
`-m <11:22:33:44:55:66>`	MAC address of VM. If MAC address changes for a TD guest, RTMR value will change and Intel TDX measurement will fail
`-q [tdvmcall | vsock]`	TD quote generation supports using tdvmcall or vsock. Choose the corresponding value to boot TD guest.
`-c <number>`	Number of vCPU. Default value is 1.
`-r <root partition>`	Root partition for direct boot, default is /dev/vda3
`-e <extra kernel command>`	Extra kernel command needed in VM boot
`-v`	Flag to enable vsock
`-d`	Flag to enable "debug=on" for GDB guest. Refer to chapter 6 Develop & Debug

-s	Flag to use serial console instead of hypervisor virtual console (HVC).
-h	Show usage help

- Direct boot TD guest via QEMU command

  This is an example of direct boot using start-qemu.sh. You need to provide the guest image and kernel image as shown. Direct boot is used by default, so it's not required to use "-b direct".

```
$ ./start-qemu.sh -i <guest image> -k <kernel binary>
```

- Grub boot TD guest via QEMU command

  This is an example of grub boot using start-qemu.sh. You need to provide the guest image and specify to use grub boot via "-b grub".

```
$ ./start-qemu.sh -i <guest image> -b grub
```

- Direct boot non-confidential guest via QEMU command

  This is an example of how to direct boot a non-TD guest. Provide the guest image and kernel image as shown. It requires using "-t efi" to boot non-confidential guest via OVMF/TDVF or "-t legacy" to boot non-confidential guest via legacy SeaBIOS.

```
$ ./start-qemu.sh -i <guest image> -k <kernel image> -t efi
$ ./start-qemu.sh -i <guest image> -k <kernel image> -t legacy
```

3.2.2 Launch via Libvirt

Libvirt is a popular orchestrator to manage the VM guest via the virsh command. Tdx-tools provides both direct boot and grub boot XML templates for TD guest at tdx-tools/doc/.

Template	Description
tdx_libvirt_direct.xml.template	TD guest direct boot
tdx_libvirt_grub.xml.template	TD guest grub boot

NOTE: The templates may vary with a different kernel version or QEMU version. Please make sure to use the correct tag of tdx-tools which matches the release version.

To create the final VM's XML from the template, update the XML template to refer to the guest image, kernel image, and OVMF binary:

- Update OVMF binary

```
<loader>/path/to/OVMF.fd</loader>
```

- Update guest image

```
<source file="/path/to/guest-image.qcow2"/>
```

- Update kernel image (This is not needed when using grub boot template)

```
<kernel>/path/to/vmlinuz-jammy</kernel>
```

Unlike QEMU, Libvirt uses the concept of a domain to manage the VM lifecycle across reboot cycle. Libvirt distinguishes between two different types of domains: transient and persistent[3].

- Transient domains only exist until the domain is shut down or when the host server is restarted.
- Persistent domains last indefinitely.

This example uses a Transient domain to start a TD guest:

```
$ virsh create tdx_libvirt_direct.xml
```

Check whether a TD guest is running with the following command. It's expected to see a TD guest running.

[3] https://wiki.libvirt.org/VM_lifecycle.html

```
$ virsh list
 Id    Name                                      State
--------------------------------------------------------------
 2     td-guest                                  running
```

Enter the TD guest console with the following command.

```
$ virsh console <TD guest name>
```

3.3 Secure Boot

Secure boot for a TD guest is almost the same as a traditional non-confidential VM. The major difference is the OMVF.fd/TDVF.fd needs to be measured into MRTD statically. Since the EFI variable is read-only in runtime with TDX guest VM, it does not permit enrolling the secure boot key into the EFI variable FV (firmware volume) via a tool such as EnrollDefaultKey at runtime. Instead, a new tool ovmfkeyenroll from tdx-tools is developed to help enroll the secure boot certificate offline before measurement.

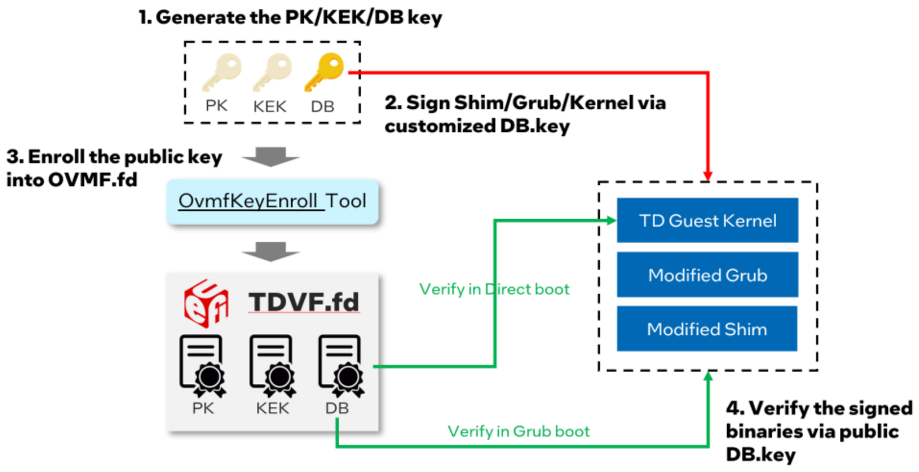

1. Generate the PK/KEK/DB key
PK
KEK
DB
2. Sign Shim/Grub/Kernel via customized DB.key
3. Enroll the public key into OVMF.fd
OvmfKeyEnroll Tool
TDVF.fd
PK
KEK
DB
Verify in Direct boot
TD Guest Kernel
Modified Grub
Modified Shim
Verify in Grub boot
4. Verify the signed binaries via public DB.key

Figure 15: Enable Secure Boot

The steps of enrolling the secure boot key are as follows:

- Step 1: Generate customized secure boot keys and certificates, instead of using a Microsoft certificate.

```
#!/bin/bash
NAME="Test"
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=$NAME PK/" -keyout PK.key \
-out PK.crt -days 3650 -nodes -sha256
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=$NAME KEK/" -keyout KEK.key \
-out KEK.crt -days 3650 -nodes -sha256
```

```
openssl req -new -x509 -newkey rsa:2048 -subj "/CN=$NAME DB/" -keyout DB.key \
-out DB.crt -days 3650 -nodes -sha256
openssl x509 -in PK.crt -out PK.cer -outform DER
openssl x509 -in KEK.crt -out KEK.cer -outform DER
openssl x509 -in DB.crt -out DB.cer -outform DER
GUID=$(python3 -c 'import uuid; print(str(uuid.uuid1()))')
echo $GUID > myGUID.txt
chmod 0600 *.key
```

Refer to the following to learn more about the different types of digital certificates.

- Managing EFI Boot Loaders for Linux: Controlling Secure Boot
- UEFI Specification

- Step 2: Build and install ovmfkeyenroll tool, Refer to source
- Step 3: Enroll key into OVMF.fd (aka TDVF.fd)

```
$ ovmfkeyenroll -fd <absolute-path-to-OVMF.fd> \
                -pk <pk-key-guid> <absolute-path-to>/PK.cer \
                -kek <kek-guid> <absolute-path-to>/KEK.cer \
                -db <db-key-guid> <absolute-path-to>/DB.cer
```

NOTE: Replace GUID with the content of myGUID.txt generated above.

- Step 4: Install signing tool
  - Option 1: Build from source: sbsigntools
  - Option 2: Use RPM or DEB packages built by a third-party.

```
$ wget https://download-
ib01.fedoraproject.org/pub/fedora/linux/releases/33/Everything/
x86_64/os/Packages/s/sbsigntools-0.9.4-2.fc33.x86_64.rpm
$ sudo rpm -ihvf sbsigntools-0.9.4-2.fc33.x86_64.rpm
```

- Step 5: Extract the following components from the guest's packages
  - shimx64.efi
  - mmx64.efi
  - grubx64.efi
  - fbx64.efi
  - guest kernel binary file like vmlinuz
- Step 6: Sign Shim/Grub/Kernel with customized secure boot key

```
$ sbsign --key <path-to>/DB.key --cert <path-to>/DB.crt --output shimx64-
signed.efi shimx64.efi
$ sbsign --key <path-to>/DB.key --cert <path-to>/DB.crt --output mmx64-signed.efi
mmx64.efi
$ sbsign --key <path-to>/DB.key --cert <path-to>/DB.crt --output grubx64-
signed.efi grubx64.efi
$ sbsign --key <path-to>/DB.key --cert <path-to>/DB.crt --output fbx64-signed.efi
fbx64.efi
$ sbsign --key <path-to>/DB.key --cert <path-to>/DB.crt --output vmlinuz-signed
vmlinuz-<guest-kernel-version>
```

NOTE: if the DB.key or DB.crt file to be signed is not in the same directory then a relative address needs to be used.

The following files are created:

- shimx64-signed.efi
- mmx64-signed.efi
- grubx64-signed.efi
- fbx64-signed.efi
- vmlinuz-signed

- Step 7: Customize guest QCOW2 Image
  - Create the directories for mounting ESP and rootfs partitions:

```
$ mkdir -p workspace/efi
$ mkdir -p workspace/rootfs
```

  - Connect the QCOW2 image to `/dev/nbdx`:

```
$ sudo modprobe nbd max_part=8
$ sudo QEMU-nbd --connect=/dev/nbd0 /path/of/td-guest.qcow2
```

  - Mount ESP and rootfs

    For RHEL 8.x:

```
$ sudo mount /dev/nbd0p2 workspace/efi
$ sudo mount /dev/nbd0p3 workspace/rootfs
```

    For Ubuntu 22.04:

```
$ sudo mount /dev/nbd0p15 workspace/efi
$ sudo mount /dev/nbd0p1 workspace/rootfs
```

  - Replace the files

    For RHEL 8.x:

```
$ sudo su
$ cp /path/to/shimx64-signed.efi workspace/efi/EFI/BOOT/BOOTX64.EFI
$ cp /path/to/shimx64-signed.efi workspace/efi/EFI/redhat/shimx64.efi
$ cp /path/to/fbx64-signed.efi workspace/efi/EFI/BOOT/fbx64.efi
$ cp /path/to/mmx64-signed.efi workspace/efi/EFI/BOOT/mmx64.efi
$ cp /path/to/mmx64-signed.efi workspace/efi/EFI/redhat/mmx64.efi
$ cp /path/to/grubx64-signed.efi workspace/efi/EFI/redhat/grubx64.efi

$ # please pay attention to replace the correct version of the kernel
$ cp /path/to/vmlinuz-signed workspace/rootfs/boot/vmlinuz-<kernel-version>
```

    For Ubuntu 22.04:

```
$ sudo su
$ cp /path/to/shimx64-signed.efi workspace/efi/EFI/BOOT/BOOTX64.EFI
$ cp /path/to/shimx64-signed.efi workspace/efi/EFI/ubuntu/shimx64.efi
$ cp /path/to/fbx64-signed.efi workspace/efi/EFI/BOOT/fbx64.efi
$ cp /path/to/mmx64-signed.efi workspace/efi/EFI/BOOT/mmx64.efi
$ cp /path/to/mmx64-signed.efi workspace/efi/EFI/ubuntu/mmx64.efi
$ cp /path/to/grubx64-signed.efi workspace/efi/EFI/ubuntu/grubx64.efi
$
$ # please pay attention to replace the correct version of the kernel
$ cp /path/to/vmlinuz-signed workspace/rootfs/boot/vmlinuz-<kernel-version>
```

- Step 8: Unmount the ESP and rootfs partitions:
  For RHEL 8.x:

```
$ sudo mount /dev/nbd0p2 workspace/efi
$ sudo mount /dev/nbd0p3 workspace/rootfs
```

For Ubuntu 22.04:

```
$ sudo mount /dev/nbd0p15 workspace/efi
$ sudo mount /dev/nbd0p1 workspace/rootfs
```

- Step 9: Disconnect the QCOW2 image

```
$ sudo QEMU-nbd --disconnect /dev/nbd0
```

Use the modified OVMF.sb.fd and tdx-guest.sb.qcow2 to start a TDVM and verify whether the secure boot is enabled via the dmesg log:

```
$ dmesg | grep -i "Secure Boot"
```

It is expected to show "`Secure Boot Enabled`"

3.4 Use VirtIO Device

Within the Intel TDX guest, the drivers are the largest threat attack surface by far. They access the host-controlled PCI config space to perform MMIO and port IO. Refer to Figure 16: TDX Guest Attack Surface or detail threat analysis at [2].

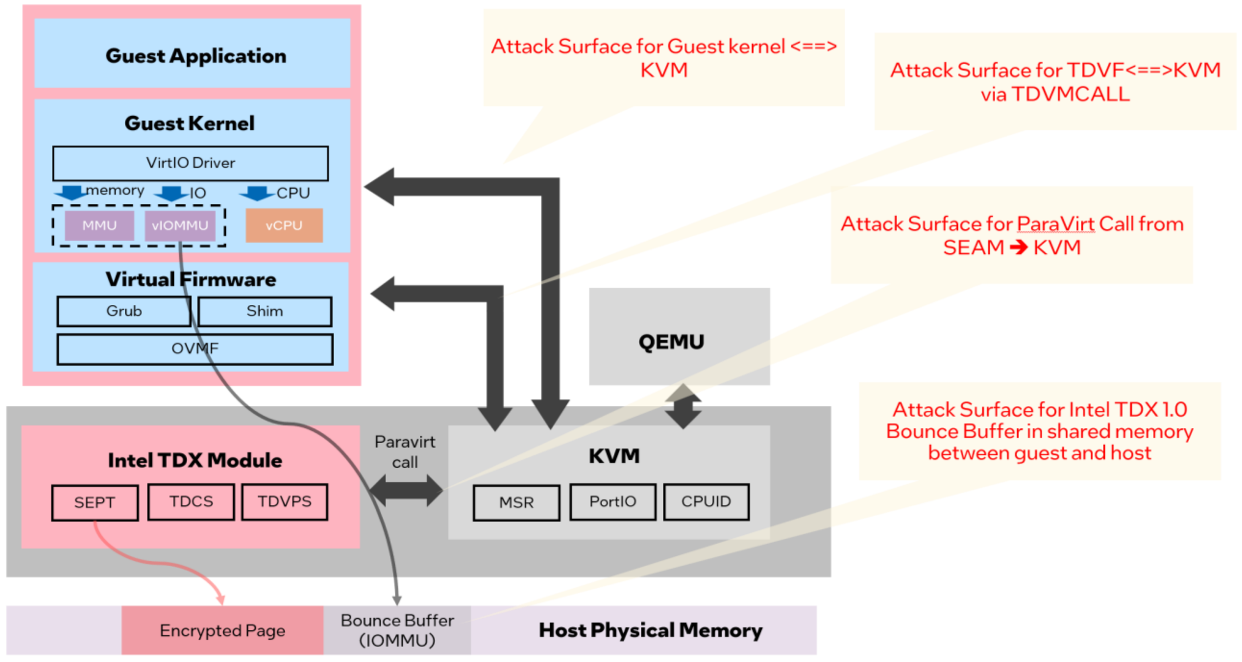

Guest Application
Guest Kernel
VirtIO Driver
memory
IO
CPU
MMU
vIOMMU
vCPU
Virtual Firmware
Grub
Shim
OVMF
Intel TDX Module
SEPT
TDCS
TDVPS
Paravirt call
QEMU
KVM
MSR
PortIO
CPUID
Encrypted Page
Bounce Buffer (IOMMU)
Host Physical Memory
Attack Surface for Guest kernel <==> KVM
Attack Surface for TDVF<==>KVM via TDVMCALL
Attack Surface for ParaVirt Call from SEAM ➔ KVM
Attack Surface for Intel TDX 1.0 Bounce Buffer in shared memory between guest and host

Figure 16: TDX Guest Attack Surface

To mitigate this risk there is a curated list of drivers that are enabled in the runtime for the TD guest kernel. By default, all PCI and ACPI bus drivers are blocked unless they are in the allow-list. The current default allow-list for the PCI bus is limited to the following VirtIO drivers:

- virtio_net
- virtio_console
- virtio_blk
- 9pnet_virtio
- virtio_vsock

Since most of the ACPI tables are not needed for an Intel TDX guest, the implemented ACPI table allow-list limits them to a small, predefined list with a possibility to pass additional tables via a command line option. The current allow-list is limited to the following tables:

- XSDT
- FACP
- DSDT
- FACS
- APIC
- SVKL

- CCEL

3.5 Full Disk Encryption

FDE (Full disk encryption) is a security method for protecting sensitive data by encrypting all data on a disk partition. In non-confidential VM, FDE is using LUKS (Linux Unified Key Setup) with a user input disk encryption key. In confidential environments like Intel TDX, to achieve zero trust, the encryption key should be retrieved from the replying party via remote attestation.

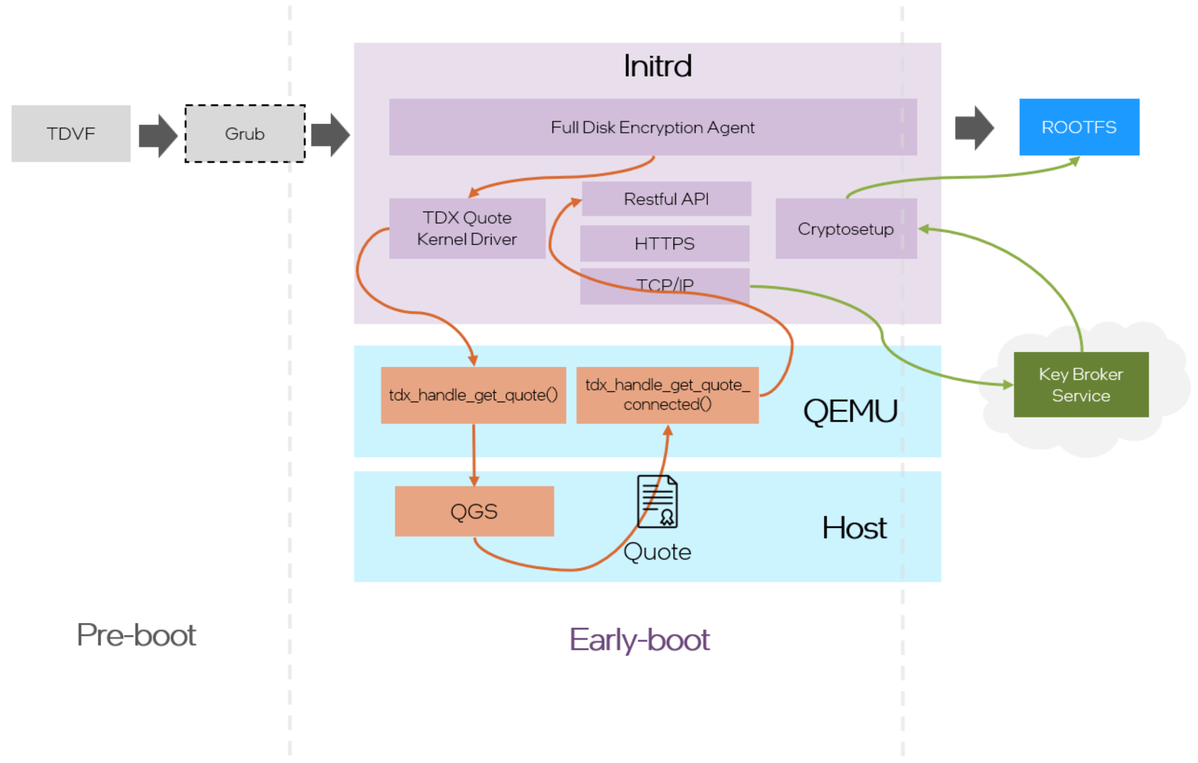


Figure 17: Full Disk Encryption in TDX Guest

The FDE can be done in OVMF at the pre-boot stage or initrd at the Linux early boot stage like. To learn more refer to the presentation "[Secure Bootloader for Confidential Computing]()".

3.5.1 Workflow

This section introduces a solution/implementation integrating FDE with Intel TDX. The workflow can be divided into 5 steps.

1. Register key and keyid from the Key Broker Service (KBS).

2. Create an encrypted guest image with key retrieved in Step 1
3. Install FDE components in the encrypted guest image
4. Enroll necessary variables into OVMF
5. Launch a TDX guest based on the encrypted guest image and the OVMF

In above step 1, the key and keyid pair should be registered in the KBS. Typically, the key will be used to encrypt the guest image, and the keyid is the identifier of the key in the KBS, which will be used in the decryption process. Given that KBS providers have different designs for their keys and keyids, it is recommended to register the pair of the key and the keyid after consulting the KBS provider.

In above step 2, it creates an encrypted guest image.

In above step 3, FDE components will be installed in the encrypted guest image. The tdx-tools provides an integrated script "`tdx-tools/attestation/full-disk-encryption/tools/image/fde-image.sh`" to complete the task. The key and the keyid is retrieved in Step 1, and the tdx-repo is built from the tdx-tools.

```
$ cd attestation/full-disk-encryption/tools/image
$ ./fde-image.sh -k ${key} -i ${keyid} -d ${tdx-repo}
```

In Step 4, several variables are enrolled in the OVMF. These variables, such as keyid, are retrieved by the fde-agent from the OVMF to help remote attestation retrieve the key from the KBS. For example, assume that the keyid is saved in a JSON file. The python script "`tdx-tools/attestation/full-disk-encryption/tools/image/enroll_vars.py`" helps enroll the data.

```
$ cd attestation/full-disk-encryption/tools/image
$ # Enroll user data
$ cat userdata.txt
{
    "keyid":"sth"
}
$ NAME="KBSUserData"
$ GUID="732284dd-70c4-472a-aa45-1ffda02caf74"
$ DATA="userdata.txt"
$ python3 tools/image/enroll_vars.py -i OVMF.fd -o OVMF.fd -n $NAME -g $GUID -d
$DATA
$ # Enroll KBS URL
$ NAME="KBSURL"
$ GUID="0d9b4a60-e0bf-4a66-b9b1-db1b98f87770"
$ DATA="url.txt"
$ python3 tools/image/enroll_vars.py -i OVMF.fd -o OVMF.fd -n $NAME -g $GUID -d
$DATA
$ # Enroll KBS Certificate
$ NAME="KBSCert"
$ GUID="d2bf05a0-f7f8-41b6-b0ff-ad1a31c34d37"
$ DATA="cert.cer"
$ python3 tools/image/enroll_vars.py -i OVMF.fd -o OVMF.fd -n $NAME -g $GUID -d
$DATA
```

In Step 5, a TDX guest is launched from the encrypted guest image. The script "tdx-tools/start-qemu.sh" can launch it. Please use the encrypted guest image and OVMF mentioned in step 3 and step 4.

```
$ OVMF_PATH=/path/to/OVMF
$ IMAGE_PATH=/path/to/image
$ start-qemu.sh \
    -b grub \
    -q tdvmcall \
    -o ${OVMF_PATH} \
    -i ${IMAGE_PATH}
```

The detail steps are described in `tdx-tools/doc/full_disk_encryption.md`.

3.5.2 Prepare Encryption Image

It is complicated to create an encrypted guest image in Step 2 and Step 3. In Step 2, an empty image is created first. The image will be partitioned into several volumes, and the root filesystem partition is encrypted with the key in actual. Then the rootfs is copy to the root filesystem partition.

In the Step 3, a binary named by the fde-agent and its related configuration need to be installed into the initrd. The parameter "cryptdevice=${root-enc}", which specifies the encrypted root partition, is appended in the kernel cmdline to enable the FDE.

More details are described in the `tdx-tools/attestation/full-disk-encryption/README.md`.

4 Measurement & Attestation

4.1 TEE, TCB, Quote

Typically, a TEE provides the evidence or measurements of its origin and current state so that the evidence can be verified by another party either programmatically or manually. It can decide whether to trust code running in the TEE. It is typically important that such evidence is signed by hardware that can be vouched for by a manufacturer, so that the party checking the evidence has strong assurances that it was not generated by malware or other unauthorized parties. [3] The remote party allows sending the secret or key to the TEE environment after successfully verifying the evidence.

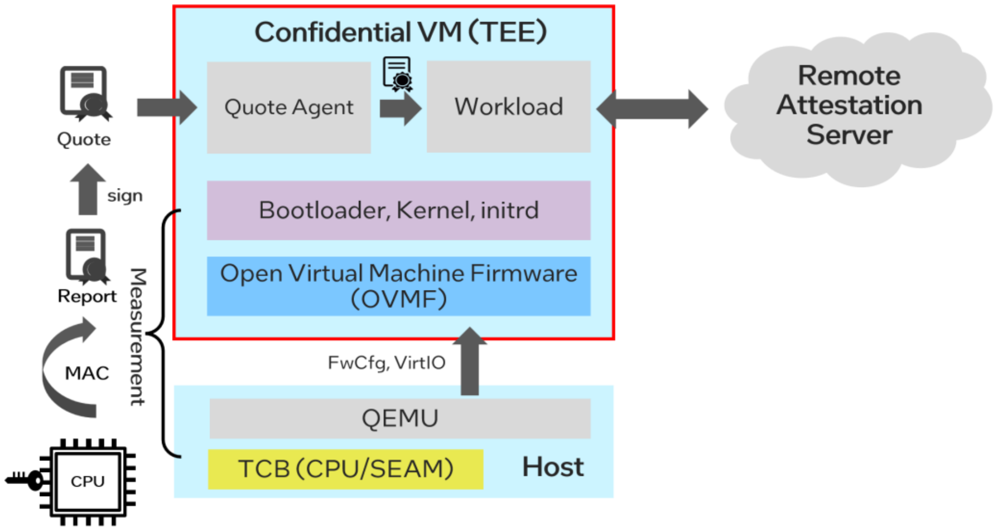


Figure 18: Measurement and Attestation for TEE

The trusted computing base (TCB) refers to all of a system's hardware, firmware, and software components that provide a secure environment. For a confidential VM, it includes hardware information such as CPU, SEAM firmware, and guest components such as OVMF, bootloader (shim/grub), and kernel. The other host software such as QEMU VMM and Orchestrator Libvirt are out of TCB.

The hash-chained measurement on TCB will be extended to some secure registers such as TPM PCR (platform configuration register). The values from several secure registers construct to a report and are finally signed to be a quote by an attestation key.

4.2 TDX Measurement

4.2.1 TD Report

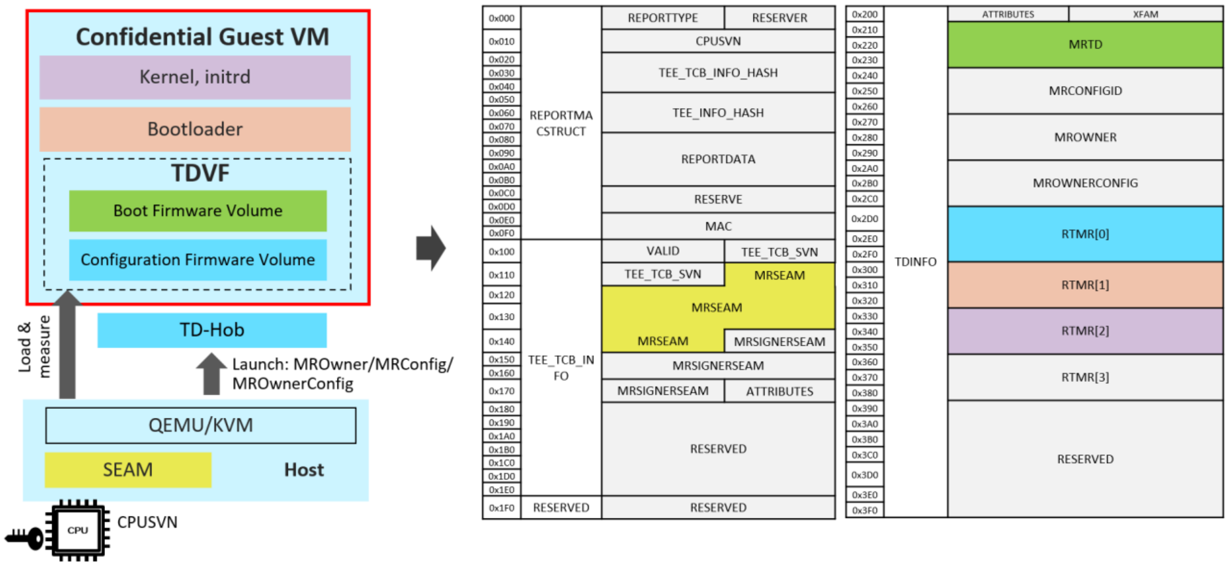


Figure 19: Intel TDX Measurement

The API TDG.MR.REPORT in the Intel TDX module creates a TDREPORT_STRUCT structure[4] containing the TD measurements, initial configuration of the TD that was locked at finalization (TDH.MR.FINALIZE), the Intel TDX module measurements, and the REPORTDATA value [1]:

- The measurement of TDX module is recorded in the field MRSEAM.
- The measurement of TDVF/OVMF is record in the field MRTD.
- The measurement of TD-Hob, ACPI is record in the RTMR [0].
- The measurement of bootloaders like grub/shim is recorded in the field RTMR [1].
- The measurement of kernel and initrd is recorded in the field RTMR [2].

NOTE: for direct boot, there is no bootloader, so the measurement of kernel is recorded in the field RTMR [1].

4.2.2 MRTD and RTMR

There are two types of measurement registers – MRTD and RTMR for Intel TDX:

[4] https://github.com/tianocore/edk2/blob/master/MdePkg/Include/IndustryStandard/Tdx.h

- MRTD (TD measurement register) provides static measurement of TD build process and the initial contents of TD
- RTMR (runtime measurement register) is an array of general-purpose measurement registers to Intel TDX software to enable measuring additional logic and data loaded into the TD at runtime. As designed, RTMR can be used by the guest TD software to measure boot process.

There are 4 RTMR registers:

Table 5: RTMR Definitions

Register	Content	Measured by
`RTMR [0]`	Static configuration (CFV); Dynamic Configuration (TD HOB, ACPI)	TDVF
`RTMR [1]`	PCI option ROM, OS loader, OS kernel, initrd, GPT, boot variable, boot parameter	TDVF
`RTMR [2]`	TD OS App	OS applications
`RTMR [3]`	Reserved	

4.2.3 Pre-Boot Measurement

The pre-boot environment before the kernel includes the TDVF/OVMF phase of the bootloader phase (shim and grub). The whole boot chain will be measured into RTMR via `EFI_CC_MEASUREMENT_PROTOCOL`[5].

[5] https://github.com/tianocore/edk2/blob/master/MdePkg/Include/Protocol/CcMeasurement.h

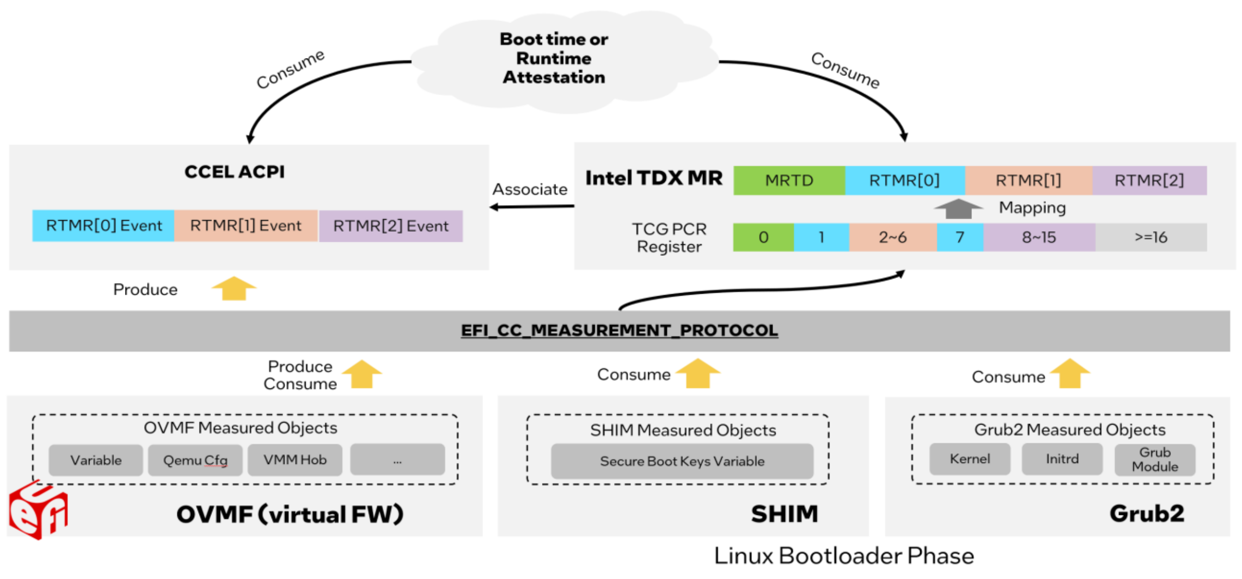


Figure 20: TD Measurement Process

Similar to TCG event log [4], `EFI_CC_MEASUREMENT_PROTOCOL` logs the events into ACPI table CCEL [6] and the measurement hash is extended to the corresponding RTMR register. The event logs in CCEL table can be replayed within a TD guest to verify the RTMR value.

4.2.4 PyTdxMeasure Tool

PyTdxMeasure in tdx-tools provides a Python library and utilities for TD measurement that can be used by tenant workload, attestation agent, or validation tools:

- Get RTMR value from TDREPORT via Linux attestation driver.
- Get the full TD event log from CCEL ACPI table.
- Verify value of RTMR by replaying event logs.

Here are the step-by-step instructions to use PyTdxMeasure:

- Install

```
$ python3 -m pip install pytdxmeasure
```

- Run
  - Get Event Log.

```
$ ./tdx_eventlogs
```

[6] https://uefi.org/specs/ACPI/6.5/05_ACPI_Software_Programming_Model.html#cc-event-log-acpi-table

Refer to the example outputs at measurement log for grub boot and measurement log for direct boot

- Get `TDREPORT`, which includes value of `RTMR`.

```
$ ./tdx_tdreport
```

- Verify `RTMR`.

```
$ ./tdx_verify_rtmr
```

The tool will compare `RTMR` value from `TDREPORT` and `RTMR` value replayed via event log. The two values are expected to be identical, which means the measured contents are not tampered with.

4.2.5 Linux Runtime Measurement

Integrity Measurement Architecture (IMA) is the Linux kernel integrity subsystem to detect if files have been accidentally or maliciously altered, both remotely and locally. Currently IMA maintains the runtime measurement list if anchored in a hardware Trusted Platform Module (TPM) to make the measured hashes of files immutable. It also supports the appraise mechanism to enforce local file integrity by appraising the measurement against a "good" value stored as an extended attribute.

Extra kernel changes have been introduced to enable IMA in TD guest and maintain the runtime measurement list inside RTMR [2].

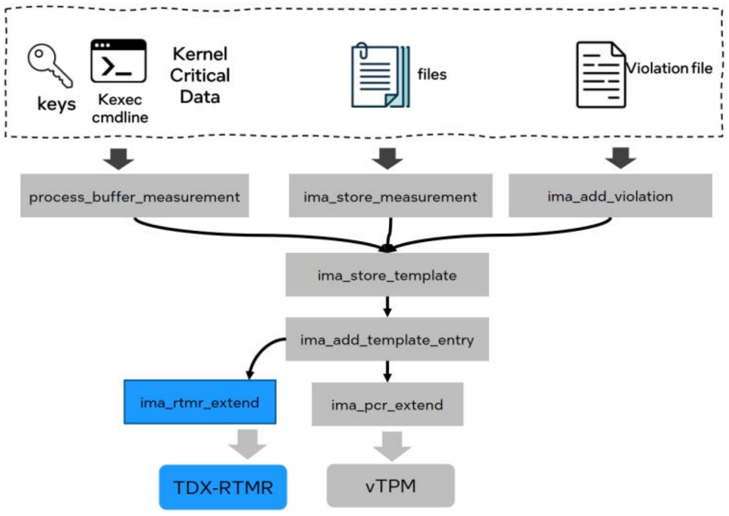


Figure 21: Enable IMA extend hash to RTMR

Different configurations (kernel command line) can be applied to define the scope to be measured. Available options include:

- "`ima_hash=sha384`": Enable measurement against boot aggregates, which covers firmware, boot loader, kernel command line and etc.
- "`ima_hash=sha384 ima_policy=critical_data`": Enable measurements against boot aggregates and kernel integrity critical data.
- "`ima_hash=sha384 ima_policy=tcb`": Enable measurements against all programs executed, files mmap'd for execution, and all files opened with the read mode bit set by either the effective uid (euid=0) or uid=0.

Custom policies can be set by user to define the scope to be measured. For more details, please refer to the IMA documentations.

Below are some sample instructions to enable and verify this feature in a TD guest:

- Sample configuration to start up the TD VM

```
$ ./start-qemu.sh -k <path-to-kernel> -i <path-to-image> -e
"ima_hash=sha384 ima_policy=critical_data"
```

- Run
  - Get IMA measurement count.

```
$ cat /sys/kernel/security/integrity/ima/runtime_measurements_count
```

  - Get full IMA measurement list stored inside the kernel securityfs.

```
$ cat /sys/kernel/security/integrity/ima/ascii_runtime_measurements
```

  - Verify RTMR within TDREPORT by using the PyTdxMeasure Tool.

```
$ cd tdx-tools/attestation/pytdxmeasure
$ sudo ./tdx_tdreport
```

User can find the measurements extended in RTMR [2] inside the TDREPORT. TPM PCR Calculator (available in Microsoft Store) can be used to replay the result with the ASCII measurements that fetched inside kernel security FS.

4.3 Attestation

4.3.1 Overview

Intel TDX remote attestation demonstrates applications that are running securely on a given trusted environment (TD guest) to a relying party. This increases the confidence of a remote party that the software is running inside a TD on a genuine Intel TDX system at a given security level, which is also referenced as the TCB version. The TDX attestation reuses Intel SGX infrastructure to provide attestation to a given measurement. It is based on TD Quote, which is the signed TD Report in TD Quoting Enclave [1].

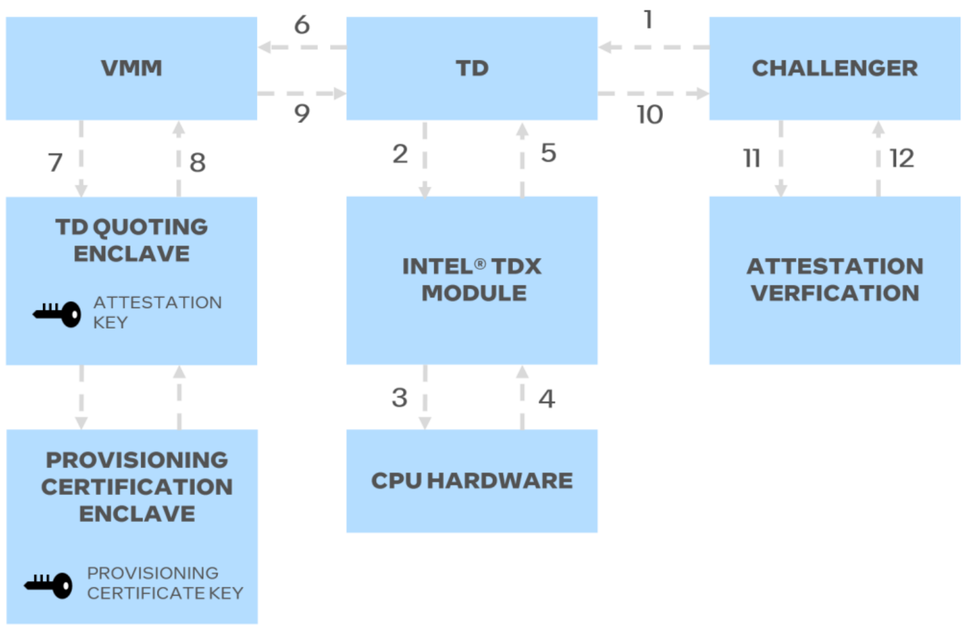


Figure 22: Intel TDX Attestation Flow

- Step 1: A TD receives an attestation request from an off-platform challenger.
- Step 2: The TD then requests an Intel TDX module to provide the TD a report
- Step 3,4: The Intel TDX module invokes the SEAMREPORT instruction to request the CPU generate a Report structure, including the TD-provided data, the measurements of the TD as maintained by the module, and SVNs (security version number) of all elements in the TDX TCB.
- Step 5, 6: The TD requests the VMM convert the report into a Quote for remote attestation.
- Step 7, 8, 9: The TD-quoting enclave then verifies the MAC on the report using EVERIFYREPORT2 and converts the report, if verified, into a Quote by signing the report using the TD's asymmetric-attestation key.
- Step 10: The Quote is returned to the challenger.
- Step 11, 12: The challenger uses an attestation-verification service to perform quote verification.

The Linux Stack for Intel TDX provides end-to-end Intel TDX attestation capability by integrating the Intel® Software Guard Extensions Data Center Attestation

Primitives[7] (Intel® SGX DCAP). In this section, it will introduce how to run Intel TDX remote attestation.

4.3.2 Set Up DCAP Repo (Host)

Before running the steps, download DCAP from https://download.01.org/intel-sgx/latest/dcap-latest/linux/ based on the OS distro.

This example shows how to set up the package repository on an Intel TDX host with either Ubuntu 22.04 or RHEL 8.x.

Get the latest instruction from https://download.01.org/intel-sgx/latest/dcap-latest/linux/docs/ or https://github.com/intel/SGXDataCenterAttestationPrimitives

1. Ubuntu 22.04

```
$ # Switch to root user
$ sudo su
$
$ tar zxvf <sgx_debian_local_repo file name>.tar.gz
$ mv sgx_debian_local_repo /srv/sgx_debian_local_repo
$
$ # Set up local Debian repository
$ cat <<EOF >> /etc/apt/sources.list.d/sgx_debian_local_repo.list
deb [trusted=yes arch=amd64] file:/srv/sgx_debian_local_repo jammy main
EOF
$
$ apt update
$ apt install -y gcc make tar

 # Install latest nodejs, version 18 shown below is an example
$ curl -sL https://deb.nodesource.com/setup_18.x -o nodesource_setup.sh
$ bash nodesource_setup.sh sudo apt-get install -y nodejs
```

2. RHEL 8.x

```
$ # Switch to root user
$ sudo su
$
$ cd /srv/
$ tar zxvf <sgx_rpm_local_repo file name>.tar.gz
$ mv sgx_rpm_local_repo /srv/sgx_rpm_local_repo
$
$ # Set up local RPM repository
$ cat <<EOF >> /etc/yum.repos.d/tdx-attestation.repo
[tdx-attestation-local]
name=tdx-attestation-local
baseurl=file:///srv/sgx_rpm_local_repo
enabled=1
```

[7] https://github.com/intel/SGXDataCenterAttestationPrimitives

```
gpgcheck=0
module_hotfixes=true
EOF
$
$ dnf check-update
$ dnf install -y gcc make tar sudo dnf module reset nodejs
$
$ # Install latest nodejs, version 18 shown below is an example
$ dnf module install nodejs:18
```

4.3.3 Set Up PCCS

Intel provides a reference provisioning certification caching service (PCCS) to enable Intel SGX attestation runtime workloads without a dependence on the Intel services. PCCS is a reference caching server to allow a CSP or a data center to cache PCK Certificates and other endorsements from the Intel® Software Guard Extensions Provisioning Certification Service (Intel® SGX Provisioning Certification Service) in their local network. You'll need to set up PCCS for remote attestation purpose.

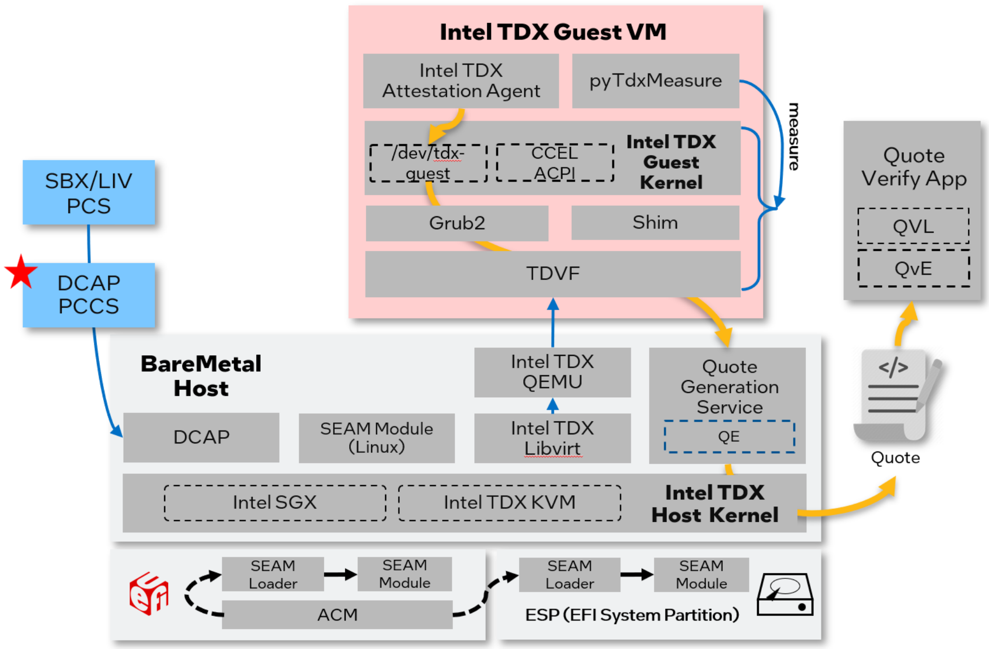


Figure 23: Setup PCCS

1. Obtain a provisioning API key for to enable the PCCS RESTful API.

Go to https://api.portal.trustedservices.intel.com/provisioning-certification and click 'Subscribe'. An API key will be generated. Be sure to keep the API key secure and also for future use.

2. Install the package sgx-dcap-pccs

   - Ubuntu 22.04

```
$ # Switch to root user for installation
$ sudo su
$
$ apt update
$ apt install -y --no-install-recommends sgx-dcap-pccs
$ cd /opt/intel/sgx-dcap-pccs
$ sudo -u pccs ./install.sh
```

   - RHEL 8.x

```
$ # Switch to root user for installation
$ sudo su
$
$ dnf install -y sgx-dcap-pccs
$ cd /opt/intel/sgx-dcap-pccs
$ sudo -u pccs ./install.sh
```

During the installation, when prompted for the API key and password, use the API key from the previous step. Other steps can accept a default value when prompted.
After the installation is completes successfully, make sure the PCCS is configured to use the v4 API. Look for "uri" in the configuration file /opt/intel/sgx-dcap-pccs/config/default.json:

```
"uri": "https://api.trustedservices.intel.com/sgx/certification/v4/"
```

3. Restart the PCCS

```
$ sudo systemctl restart pccs
```

Note: Please Delete the previously created database before restarting the PCCS service.

```
$ sudo rm -rf /opt/intel/sgx-dcap-pccs/pckcache.db
$ sudo systemctl restart pccs
$ sudo systemctl status pccs
```

4. Check the PCCS service log

```
$ sudo journalctl -u pccs -f
```

4.3.4 Set Up DCAP on Host

This section introduces the installation of Quote Generation Service from DCAP and performs Intel SGX platform registration.

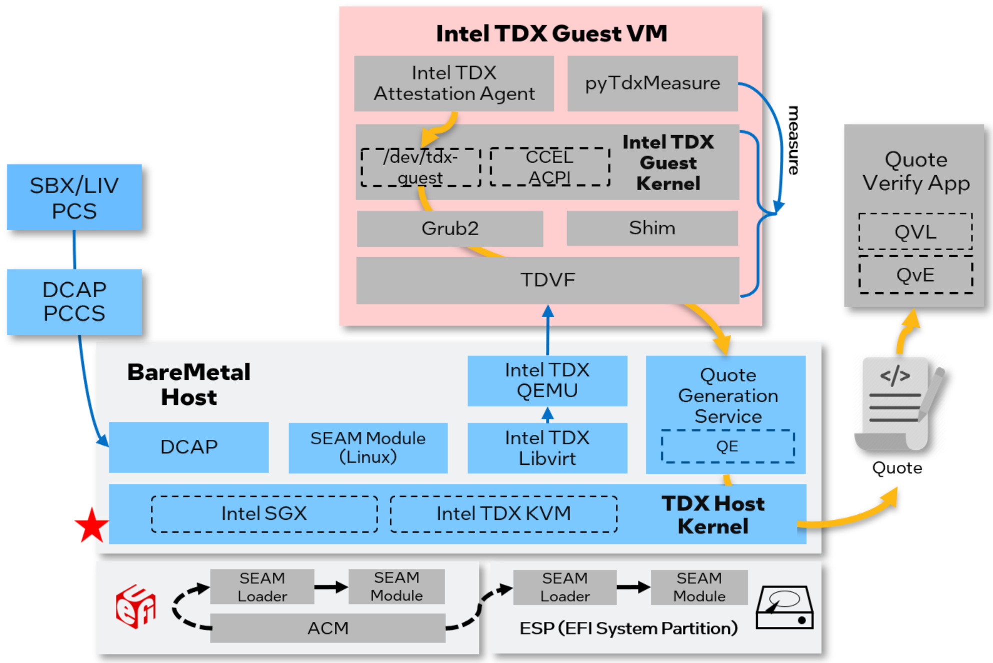


Figure 24: Set up DCAP software on the TDX host

1. Install the Intel® Software Guard Extensions SDK for Linux* OS (Intel® SGX SDK for Linux* OS) to the folder /opt/intel/
   - Ubuntu 22.04 host

```
$ sudo ./sgx_linux_x64_sdk_2.18.100.4-u2204.bin
```

   - RHEL 8.x host.

```
$ sudo ./sgx_linux_x64_sdk_2.18.100.4.bin
```

2. Install QGS and QPL packages on the host.
   - Ubuntu 22.04 host

```
$ sudo apt install -y -no-install-recommends tdx-qgs libsgx-dcap-default-qpl
```

   - RHEL 8.x host

```
$ sudo dnf install -y tdx-qgs libsgx-dcap-default-qpl
```

Modify the configuration file: /etc/sgx_default_qcnl.conf to add the following.

```
// PCCS server address
“pccs_url”: “https://<PCCS_IP>:8081/sgx/certification/v4/”,
// To accept insecure HTTPS certificate depends on PCCS Server’s configuration,
// set below option to false
“use_secure_cert”: false
```

3. Install PCKIDRetrievalTool

   - Ubuntu 22.04 host

```
$ sudo apt install -y sgx-pck-id-retrieval-tool
```

   - For RHEL 8.* host, run below commands:

```
$ sudo dnf install -y sgx-pck-id-retrieval-tool
```

NOTE: the reported version of the PCKIDRetrievalTool may be different.

4. Modify the configuration file `/opt/intel/sgx-pck-id-retrieval-tool/network_setting.conf` with the following content.

```
PCCS_URL=https://<PCCS_IP>:8081/sgx/certification/v4/platforms
# if using localhost as pccs
# PCCS_URL=https://localhost:8081/sgx/certification/v4/platforms
USE_SECURE_CERT=FALSE
```

5. Do SGX platform Registration via PCKIDRetrievalTool

```
$ sudo sh -c “./PCKIDRetrievalTool”
```

   The expected response is as follows. The reported version may be different.

```
Intel® Software Guard Extensions PCK Cert ID Retrieval Tool Version 1.14.100.3
Registration status has been set to completed status. Pckid_retrieval.csv has been
generated successfully!
```

NOTE: If it returns a message like “Platform Manifest not available”, you may need to perform SGX Factory Reset in BIOS and run PCKIDRetrievalTool again.

4.3.5 Generate Quote

This section introduces the quote generation steps including launching TDX with quote generation support and generating a quote within the TDX guest.

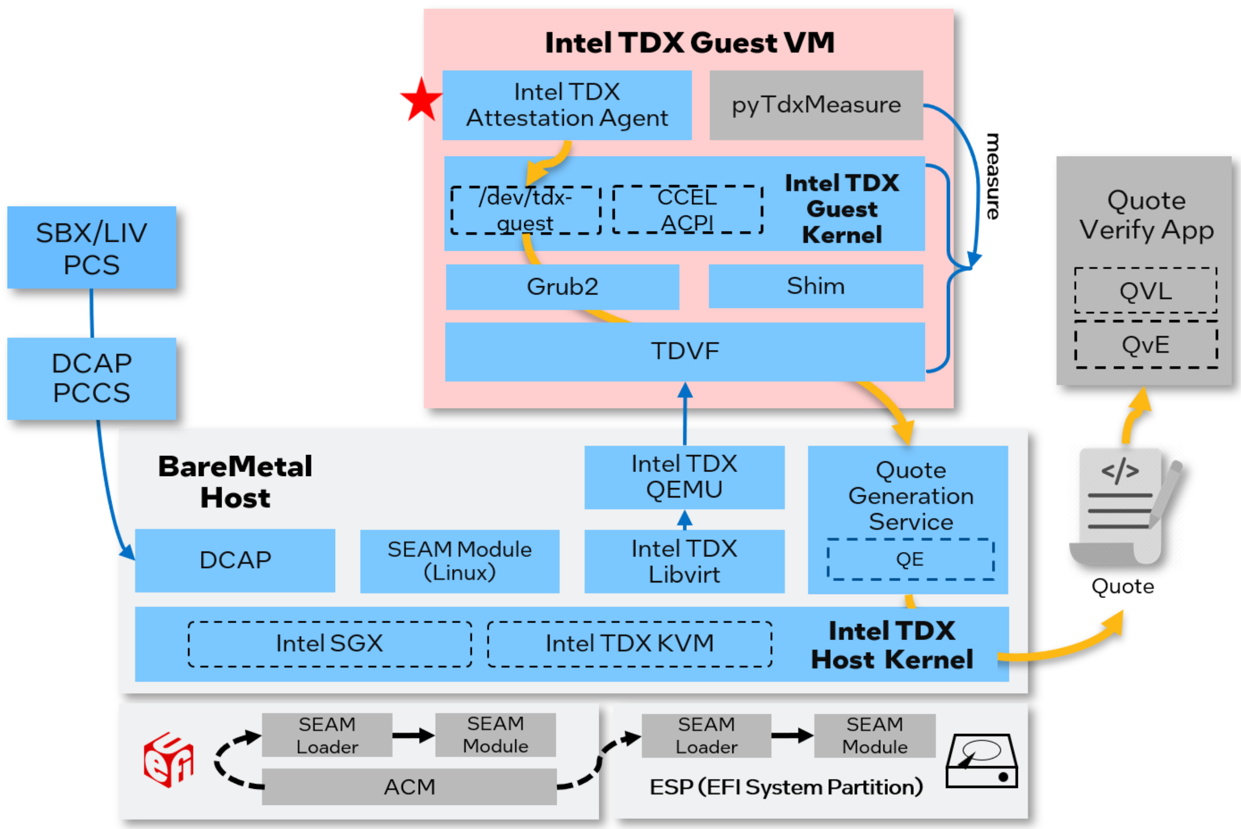
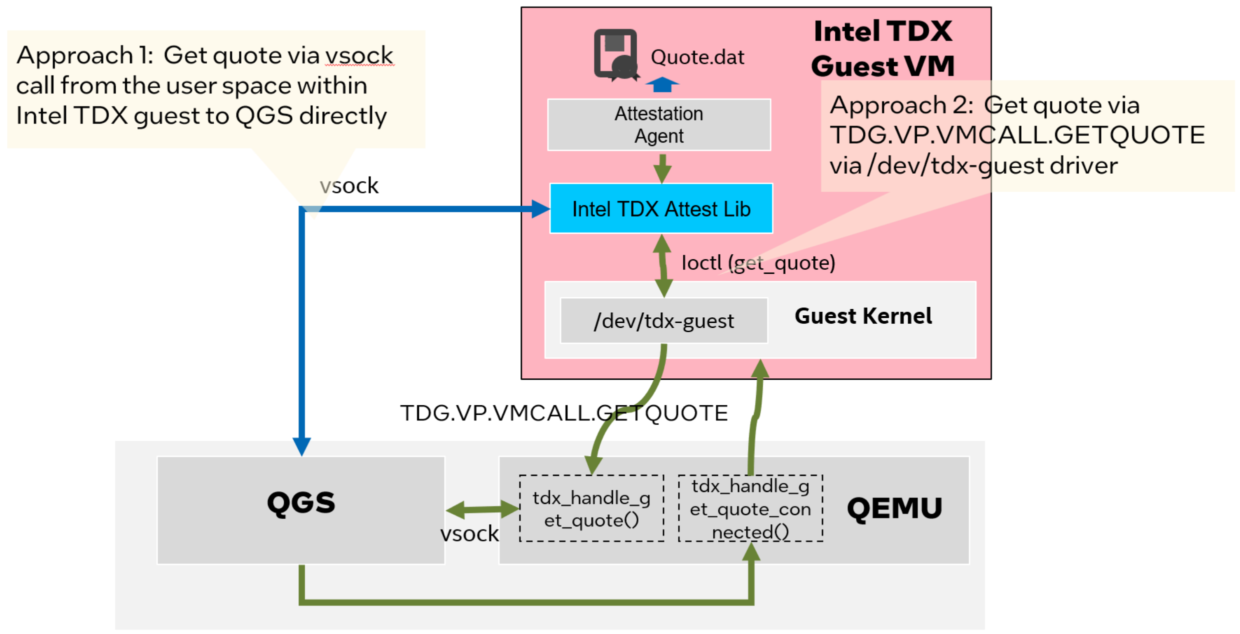


Figure 25: Quote Generation

4.3.5.1 Launch TD with Quote Generation Support

There are two ways to run quote generation:



Approach 1: Get quote via vsock call from the user space within Intel TDX guest to QGS directly
vsock
Quote.dat
Intel TDX Guest VM
Attestation Agent
Approach 2: Get quote via TDG.VP.VMCALL.GETQUOTE via /dev/tdx-guest driver
Intel TDX Attest Lib
Ioctl (get_quote)
/dev/tdx-guest
Guest Kernel
TDG.VP.VMCALL.GETQUOTE
QGS
vsock
tdx_handle_get_quote()
tdx_handle_get_quote_connected()
QEMU

Figure 26: Approaches to Generate Intel TDX Quote

- Approach 1: Get quote via vsock call from the user space within TD guest to QGS directly
    - If launched via QEMU, add the following parameter.

```
-device vhost-vsock-pci,guest-cid=3
```

    - If using start-qemu.sh to indicate getting quote via vsock.

```
$ ./start-qemu.sh -i <guest image> -k <guest kernel> -q vsock
```

    - If launched via libvirt, add the following fields in XML

```
<vsock model='virtio'>
<cid auto='yes' address='3'/>
<address type='pci' domain='0x0000' bus='0x05' slot='0x00' function='0x0'/>
</vsock>
```

- Approach 2: Get quote via TDG.VP.VMCALL.GETQUOTE
    - If launched via QEMU, add "quote-generation-service=vsock:2:4050" in parameter -object

```
-object tdx-guest,sept-ve-disable,id=tdx,quote-generation-service=vsock:2:4050
```

    - If using start-qemu.sh to indicate getting quote via tdvmcall.

```
$ ./start-qemu.sh -i <guest image> -k <guest kernel> -q tdvmcall
```

    - If launched via libvirt, add following fields in XML

```
<launchSecurity type='tdx'>
......
<Quote-Generation-Service>vsock:2:4050</Quote-Generation-Service>
</launchSecurity>
```

- Within a TD guest, create a file at /etc/tdx-attest.conf with the following content:

```
port=4050
```

4.3.5.2 Generate Quote within Intel TDX Guest

1. Set up the package repository in TD guest. Refer to Section 4.3.2: Set Up DCAP .

- Install libtdx-attest, libtdx-attest-dev
    - For Ubuntu 22.04

```
$ sudo apt install -y libtdx-attest libtdx-attest-dev
```

    - For RHEL 8.x

```
$ sudo dnf install -y libtdx-attest libtdx-attest-devel
```

2. Build quote generation sample

```
$ cd /opt/intel/tdx-quote-generation-sample/
$ make clean
$ make
```

3. Use test_tdx_attest to generate a quote.dat.

```
$ sudo ./test_tdx_attest
```

4.3.6 Verify Quote

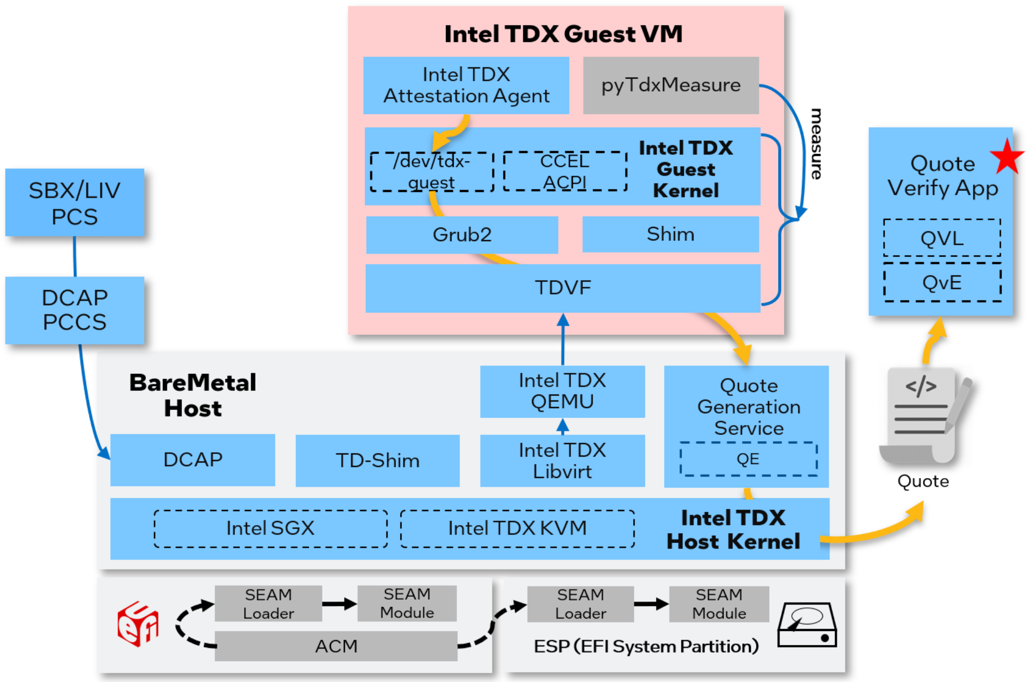


Figure 27: Verify Quote

After the Quote is generated, you can use a sample Quote verification application to verify the Quote.

1. Install the Quote verification libraries:

   - For Ubuntu 22.04

```
$ sudo apt install -y libsgx-dcap-quote-verify
$ sudo apt install -y libsgx-dcap-quote-verify-dev
$ sudo apt install -y libsgx-ae-qve
```

   - For RHEL 8.x

```
$ sudo dnf install -y libsgx-dcap-quote-verify
$ sudo dnf install -y libsgx-dcap-quote-verify-devel
$ sudo dnf install -y libsgx-ae-qve
```

2. Copy quote.dat from TDVM

   Use scp or virt_copy_out to copy the quote from the TDVM.

```
$ virt-copy-out -a <image_name> <directory_in_TDVM_contains_quote.dat>
<a_host_directory>
```

NOTE: Terminate the TDVM before using virt_copy_out to copy out the quote.dat.

3. Build and run sample application verifying the generated quote located at <PATH>:

```
$ git clone https://github.com/intel/SGXDataCenterAttestationPrimitives.git
$ cd SGXDataCenterAttestationPrimitives/SampleCode/QuoteVerificationSample
$ make SGX_DEBUG=1
$ ./app -quote <PATH>/quote.dat
```

4.4 Use Intel Project Amber

Intel Project Amber is Intel's first step in creating a new multi-cloud, multi-TEE service for third-party attestation to help drive forward adoption of confidential computing for the broader industry. Please refer to Project Amber for more details. This section introduces how to install the Amber client to access services.

4.4.1 Overview

Intel® Project Amber Go Client Library[8] is a Go library for integrating with the Intel® Project Amber V1 API. A beta version Intel® Project Amber Go TDX CLI[9] (amber-cli) is also provided in this library repository. This amber-cli provides basic functionalities:

- Create RSA key pair
- Get an Amber signed token
- Get a TD Quote with nonce and user data
- Decrypting an encrypted blob

This section introduces the installation and example usages.

[8] https://github.com/intel/amber-client
[9] https://github.com/intel/amber-client/tree/main/amber-cli-tdx

4.4.2 Installation

A build script is provided by tdx-tools. Refer to 2.4.1 Build Packages to build packages and 2.5 Install IaaS Host to set up the repository.

NOTE: This package is only for TD guest's attestation environment.

- Install the Amber Client package and its dependency.
  - For Ubuntu 22.04

```
$ sudo apt install -y libtdx-attest amber-cli
```

  - For RHEL 8.x

```
$ sudo dnf install libtdx-attest intel-mvp-amber-cli
```

4.4.3 Example Usage

To get the TD Quote, a nonce and user data can be used as input parameter.

- Get a TD Quote

```
$ amber-cli quote
```

- The nonce is a value constructed to be unique to a particular message to prevent replay attacks, user can get a nonce from Amber attestation service or create random data as a nonce. To get a TD Quote with the nonce by following step, the hash of the nonce will be put into TD report in the TD Quote.

```
$ amber-cli quote --nonce <base64 encoded nonce>
```

- Get a TD Quote with nonce and user data

```
$ amber-cli quote --nonce <base64 encoded nonce> --user-data <base64 encoded
userdata>
```

To get an Amber signed token, the `AMBER_URL` and `AMBER_API_KEY` is needed, please contact Intel® Project Amber team to get them.

- Export environment variables

```
$ export AMBER_URL=<amber api url>
$ export AMBER_API_KEY=<amber attestation api key>
```

- Create RSA key pair

```
$ amber-cli create-key-pair --key-path <private key file path>
```

- Get an Amber signed token

```
$ amber-cli token
```

- Get an Amber signed token with user data and policy-ids

```
$ amber-cli token --user-data <base64 encoded userdata> --policy-ids <comma
separated amber attestation policy ids>
```

The amber-cli provides a feature to decrypt an encrypted blob. The encrypted blob should be encoded by base64.

```
$ amber-cli decrypt --key-path <private key file path> --in <base64 encoded
encrypted blob> --out <output file path>
```

5 Validation

5.1 Overview

The Linux Stack for Intel TDX provides end to end TDX capability across diverse infrastructures like hypervisor and Kubernetes within hypervisor.

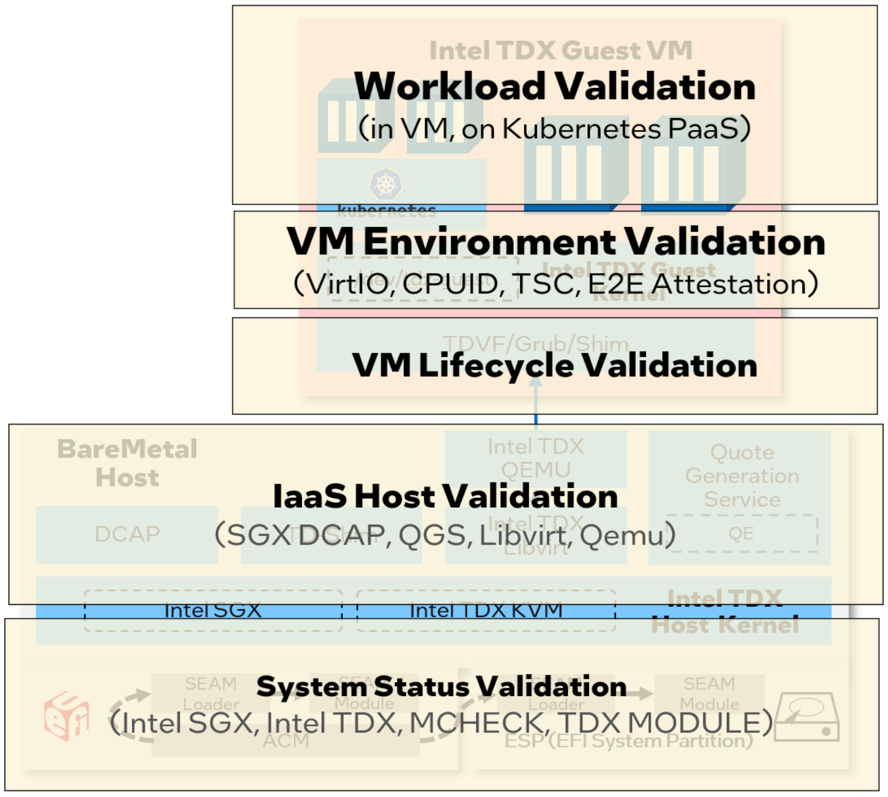


Figure 28: Intel TDX E2E Full Stack Validation

The end-to-end validation of Linux Stack for Intel TDX covers the following scopes.

Table 6: Linux Stack for Intel TDX Validations

Validation	Scope	Description
System Status	IaaS	Verify the hardware and BIOS status like Intel SGX, Intel TME-MK, Intel TDX, Intel TDX module, etc.
IaaS Host	IaaS	Verify the functionality of IaaS components like platform registration, QGS service, libvirt and QEMU configurations
VM Lifecycle	PaaS	Diverse boot types for TD VM guest, pre-boot environment measurement, etc.

VM Environment	PaaS	CPUID, TSC, VirtIO devices, etc.
Workload	PaaS	Container workloads run in TD guest or the Kubernetes cluster within TD guest

To support complex validation and automation scenarios, the pyCloudStack framework is designed to support the scopes mentioned in Table 6: Linux Stack for Intel TDX Validations.

5.2 PyCloudStack

5.2.1 Overview

PyCloudStack abstracts the common objects, operations, and resources for diverse cloud architectures. It supports hypervisor stacks based on libvirt or direct QEMU commands, container stacks orchestrated by Kubernetes or directly by docker, and supports running on local or remote IaaS hosts. It can be used to create an advanced deployment CI/CD operator via a Python plugin for ansible, end-to-end validation, framework with customized components and configurations in a full vertical stack.

The overall architecture diagram is illustrated as below:

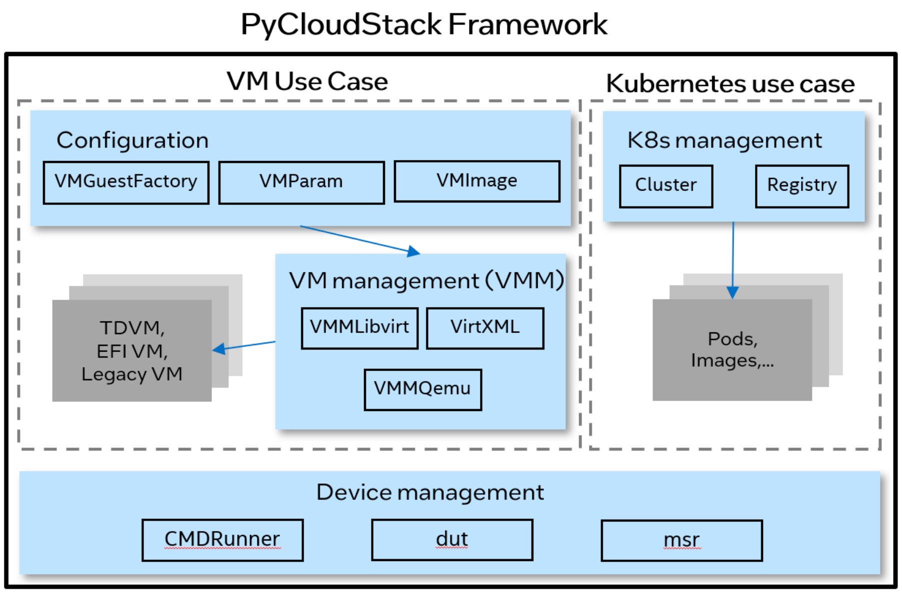


Figure 29: PyCloudStack Framework

The framework supports scenarios for VM management: via QEMU direct or via libvirt:

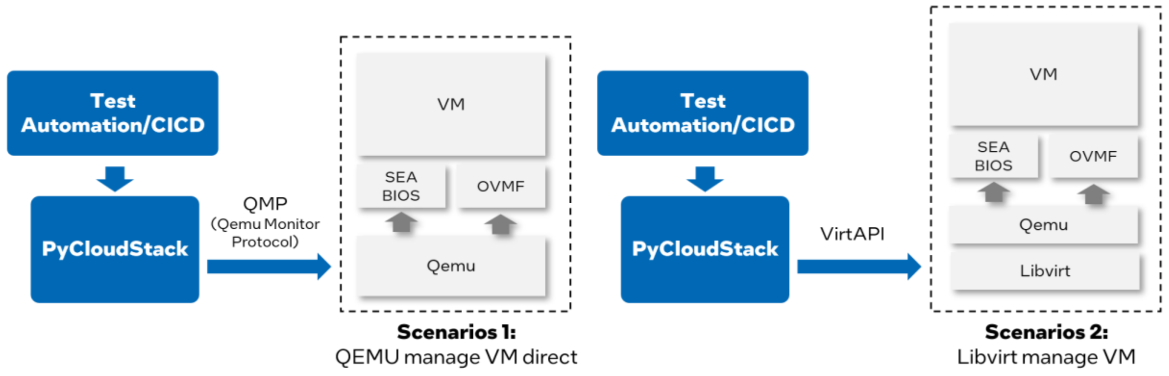


Figure 30: Validation Scenarios for VMM and Libvirt

- Scenario 1: QEMU managed VM directly via QMP (QEMU monitor protocol)[10]
- Scenario 2: Libvirt managed VM via VirtAPI[11]

The framework abstracts the common operations for host, virtual machine, kubernetes, and container:

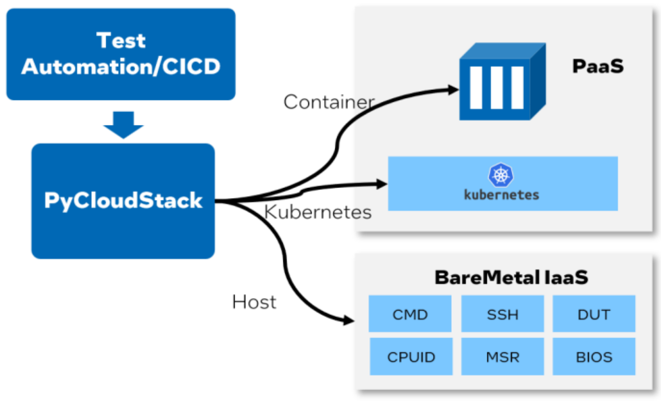


Figure 31: Abstract Common Operations for Cloud Stack

Below are additional details for the VM use case.

- `VMGuestFactory` is designed to communicate and handle VM configurations with test cases. For example, the size of a virtual machine can be specified by indicating how many CPUs and how much memory are required. `VMGuestFactory` usually works with `VMParam` and `VMImage`.
  - `VMParam` operator provides predefined VM parameters for typical configuration. It also provides the capability for you to customize VM parameters.

[10] https://wiki.QEMU.org/Documentation/QMP
[11] https://github.com/virtapi/virtapi

    - `VMImage` is designed to manage guest images for guest VMs so that multiple guest distros can be supported. You can customize guest images based on test requirement.
- `VMM` operators are responsible for VM lifecycle management using given configuration. VMM operator includes `VMMLibvirt` and `VMMQEMU`. `VMMLibvirt` needs to work together with "`virtXML`" operator.
- `virtXML` is responsible for Libvirt XML template management. It helps you to customize XML template for VMs.

For Kubernetes use case:

- `cluster` operator is designed to implement Kubernetes object management. `Registry` is used to manage container images. With them working together, you can create Kubernetes objects, such as deployment and service. Then cloud workload can run in a Kubernetes cluster.

There are also some other common operators for Device management at the bottom of the diagram.

- `CMDRunner` is designed to run commands on local host or remote targets via ssh connection.
- `DUT` is designed to manage devices under test, such as CPU frequency of host.
- `MSR` operator provides methods to read and write register.

Finally, with the PyCloudStack framework, functionality, stability, performance, and interoperability tests are well supported.

5.2.2 Installation

PyCloudStack has been uploaded to PyPI. Install PyCloudStack to the host via the following command.

```
$ pip3 install pycloudstack
```

5.2.3 Example

Most of automation tests in the tdx-tools repo are based on the PyCloudStack framework. Below are several examples:

- Example 1: Operate VM via Libvirt

```
from pycloudstack.vmguest import VMGuestFactory
from pycloudstack.vmparam import VM_STATE_SHUTDOWN, VM_STATE_RUNNING,
VM_STATE_PAUSE, VM_TYPE_TD

vm_factory = VMGuestFactory(vm_image, vm_kernel)

LOG.info("Create TD guest")
inst = vm_factory.new_vm(VM_TYPE_TD, auto_start=True)
inst.wait_for_ssh_ready()

LOG.info("Suspend TD guest")
inst.suspend()
ret = inst.wait_for_state(VM_STATE_PAUSE)
assert ret, "Suspend timeout"

LOG.info("Resume TD guest")
inst.resume()
ret = inst.wait_for_state(VM_STATE_RUNNING)
assert ret, "Resume timeout"
```

- Example 2: Customize the VM

```
import logging
import psutil
# Get host total cores and sockets, assign 80% vcpu and 80% memory to vm
total_core = psutil.cpu_count()
cores = int(total_core * 0.4)
memsize = int(psutil.virtual_memory().available / 1000 * 0.8)
vmspec = VMSpec(sockets=2, cores=cores, memsize=memsize)
inst = vm_factory.new_vm(VM_TYPE_TD, vmspec=vmspec, auto_start=True)
```

- Example 3: Run TensorFlow AI microbench boosted by AMX within TDVM

```
LOG.info("Create TD guest to test tensorflow")
td_inst = vm_factory.new_vm(vm_type, vmspec=VMSpec.model_large())

# customize the VM image
td_inst.image.inject_root_ssh_key(vm_ssh_pubkey)

# create and start VM instance
td_inst.create()
td_inst.start()
td_inst.wait_for_ssh_ready()

# It may take up to 30 minutes to complete the test
LOG.info("====== The test running may take up to 30 minutes! ======")

command = '''
cd /root/models-2.5.0 && DNNL_MAX_CPU_ISA=AVX512_CORE_AMX OMP_NUM_THREADS=16
KMP_AFFINITY=granularity=fine,verbose,compact
python3 ./benchmarks/launch_benchmark.py
    --model-name dien  --mode inference  --precision bfloat16
    --framework tensorflow --data-location /root/dien
    --exact-max-length=100 --num-inter-threads 1  --num-intra-threads 16
    --batch-size 8 --graph-type=static
    --in-graph /root/dien_fp32_static_rnn_graph.pb
    --benchmark-only --verbose --
     '''
```

```
runner = td_inst.ssh_run(command.split(), vm_ssh_key)
assert runner.retcode == 0, "Failed to execute remote command"

# throughput should not be 0
patt_ok = r'Approximate accelerator performance in recommendations/second is
(\d*.\d*)'
match = re.search(patt_ok, '\n'.join(runner.stdout))
assert match is not None
images_per_s = match.group(1)
LOG.info('Throughput: %s recommendations/s', images_per_s)
assert float(images_per_s) > 0
```

5.3 Intel TDX Tests

Intel TDX tests from tdx-tools are designed to cover basic acceptance tests, functionality, workload, and environment tests for Intel TDX. It also provides interoperability tests by using AMX in an Intel TDX guest VM.

NOTE: The tests implementation depends on the PyCloudStack framework. The test execution must be on an Intel TDX-enabled Linux platform with an Intel TDX-enabled kernel with QEMU and Libvirt installed.

NOTE: Please make sure to use the correct tag of tdx-tools which matches the release version so that the tests can work with different Intel TDX kernel and Intel TDX QEMU versions.

5.3.1 Overview

The tests can be classified into 4 categories – Lifecycle, Environment, Workload and Interoperability. Refer to the test list in the table below. Some of the tests require a customized guest image before running the test. The required prerequisites are in the next section.

Table 7: TDX Stack Tests

Category	Test case	Description
Lifecycle	test_tdvm_lifecycle.py	TD lifecycle management
	test_multiple_tdvms.py	Co-existence of multiple TDs
	test_vm_coexists.py	Co-existence 0f TD and legacy VM
	test_max_cpu.py	Boot TD with high CPU utilization
	test_vm_shutdown_mode.py	Different shutdown modes of Libvirt
	test_acpi_reboot.py	TD ACPI reboot
	test_acpi_shutdown.py	TD ACPI shutdown
	test_vm_shutdown_qga.py	VM shutdown via QEMU guest agent

	test_vm_reboot_qga.py	VM reboot via QEMU guest agent
Environment	test_tdvm_tsc.py	TD TSC clock source and frequency
	test_tdx_guest_status.py	TDX initialization in TD guest
	test_tdx_host_status.py	Check TDX host status
	test_tdvm_network.py	Check network functions in TD
Workload	test_workload_redis.py	Redis workload running in TD
	test_workload_nginx.py	Nginx workload running in TD
Interoperability	test_amx_docker_tf.py	Run AI model with AMX in docker container on TD
	test_amx_vm_tf.py	Run AI model with AMX in TD

A full example for a redis workload test case is as follows.

```
def test_tdvm_redis(vm_factory, vm_ssh_pubkey, vm_ssh_key):
    """
    Run redis benchmark test
    Ref: https://redis.io/topics/benchmarks
    Use official docker images redis:latest
    Test Steps:
    1. start VM
    2. Run remote command "systemctl status docker" to check docker service's
status
    3. Run remote command "systemctl start docker" to force start docker service
    4. Run remote command "/root/bat-script/redis-bench.sh"
       to launch redis container and  benchmark testing
    """
    LOG.info("Create TD guest to run redis benchmark")
    td_inst = vm_factory.new_vm(VM_TYPE_TD)

    # customize the VM image
    td_inst.image.inject_root_ssh_key(vm_ssh_pubkey)
    td_inst.image.copy_in(
        os.path.join(CURR_DIR, "redis-bench.sh"), "/root/")

    # create and start VM instance
    td_inst.create()
    td_inst.start()
    td_inst.wait_for_ssh_ready()

    command_list = [
        'systemctl start docker',
        '/root/redis-bench.sh -t get,set'
    ]
    for cmd in command_list:
        LOG.debug(cmd)
        runner = td_inst.ssh_run(cmd.split(), vm_ssh_key)
        assert runner.retcode == 0, "Failed to execute remote command"
```

5.3.2 Prerequisites

A guest image is required for all the tests. Refer to "Chapter 2.4.2 Create Guest Image" to generate a basic guest image. Additional prerequisites are required for some of the tests. The first step is to start a VM using the guest image built above and go through corresponding items required by tests. The next step is to shut down the VM and use the guest image for further tests.

1. Install QEMU guest agent in guest image.

   For Ubuntu 22.04 guest image:

```
$ sudo apt-get install QEMU-guest-agent
```

   For RHEL 8.x guest image:

```
$ sudo dnf install QEMU-guest-agent
```

2. Install docker in guest image.

   For Ubuntu 22.04 guest image:

```
$ sudo apt-get install docker.io
```

   For RHEL 8.x guest image:

```
$ sudo dnf install docker
```

3. For workload tests, make sure the latest docker image is in the guest image. It needs both the docker image "nginx:latest" and "redis:latest".

```
$ docker pull nginx:latest
$ docker pull redis:latest
```

4. Install `intel-tensorflow-avx512` in guest image. Download the DIEN_bf16 model and put it under /root in the guest image.

   For ubuntu 22.04 guest image:

```
$ pip3 install intel-tensorflow-avx512==2.11.0
$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v2_5_0/dien_bf16_pretrained_opt_model.pb
```

For RHEL 8.x guest image upgrade python to python 3.8 first and then run the following commands:

```
$ pip3 install intel-tensorflow-avx512==2.11.0
$ wget https://storage.googleapis.com/intel-optimized-
tensorflow/models/v2_5_0/dien_bf16_pretrained_opt_model.pb
```

5.3.3 Setup Environment

1. Install required packages:

   If your host distro is RHEL 8.x:

```
$ sudo dnf install python3-virtualenv python3-libvirt libguestfs-devel libvirt-
devel python3-devel gcc gcc-c++
```

   If your host distro is Ubuntu 22.04:

```
$ sudo apt install python3-virtualenv python3-libvirt libguestfs-dev libvirt-dev
python3-dev net-tools
```

2. Make sure the libvirt service is started. If not, start libvirt service. If the host is Ubuntu 22.04 and AppArmor is enabled, set `security_driver = "none"` in /etc/libvirt/QEMU.conf and restart the libvirt service.

```
$ sudo systemctl status libvirtd
$ sudo systemctl restart libvirtd
```

3. Setup environment. Run the below command to setup the python environment.

```
$ cd tdx-tools/tests/
$ source setupenv.sh
```

4. Create artifacts.yaml from template

   Refer template `<tdx-tools>/tests/artifacts.yaml.template` to create `<tdx-tools>/tests/artifacts.yaml`. Update
   the source and sha256sum to indicate the location of guest image and guest kernel. See following example:

```
latest-guest-image-rhel:
  source: http://cpss-devops.sh.intel.com/download/tdx-guest/latest/td-guest-
rhel8.7.qcow2.tar.xz
  sha256sum: http://css-devops.sh.intel.com/download/tdx-guest/latest/td-guest-
rhel8.7.qcow2.tar.xz.sha256sum

latest-guest-kernel-rhel:
  source: http://css-devops.sh.intel.com/download/tdx-guest/latest/vmlinuz-rhel8.7
  sha256sum: http://css-devops.sh.intel.com/download/tdx-guest/latest/vmlinuz-
rhel8.7.sha256sum
```

```
latest-guest-image-ubuntu:
  source: http://css-devops.sh.intel.com/download/tdx-guest/latest/td-guest-
ubuntu-22.04-test.qcow2.tar.xz
  sha256sum: http://css-devops.sh.intel.com/download/tdx-guest/latest/td-guest-
ubuntu-22.04-test.qcow2.tar.xz.sha256sum

latest-guest-kernel-ubuntu:
  source: http://css-devops.sh.intel.com/download/tdx-guest/latest/vmlinuz-jammy
  sha256sum: http://css-devops.sh.intel.com/download/tdx-guest/latest/vmlinuz-
jammy.sha256sum
```

5. Generate keys

   Generate a pair of keys that will be used in test running.

```
$ ssh-keygen
```

   The keys should be named "vm_ssh_test_key" and "vm_ssh_test_key.pub" and located under tdx-tools/tests/tests/

5.3.4 Run Tests

1. Run all tests:

```
$ sudo ./run.sh -s all
$ # NOTE:
$ # “sudo” is required since some tests need root permission.
$ # The user needs to be added into the libvirt group. e.g., for user "root"
$ # please run
$ sudo usermod -aG libvirt root
```

2. Run some case modules:

```
$ ./run.sh -c <test_module1> -c <test_module2>
```

   For example, run the whole test module “test_tdvm_lifecycle.py”.

```
$ ./run.sh -c tests/test_tdvm_lifecycle.py
```

3. Run specific test cases:

```
$ ./run.sh -c <test_module1> -c <test_module1>::<test_name>
```

   For example, run the test case “test_tdvm_lifecycle_virsh_start_shutdown” in “tests/test_tdvm_lifecycle.py”

```
$ ./run.sh -c tests/test_tdvm_lifecycle.py::
test_tdvm_lifecycle_virsh_start_shutdown
```

4. User can specify guest image OS type with “-g”. Currently “rhel” and “ubuntu” are supported. RHEL 8.x guest image will be used by default if “-g” is not specified.

   For example, run all the tests using an Ubuntu 22.04 guest image.

```
$ sudo ./run.sh -g ubuntu -s all
```

6 Develop & Debug

6.1 Override the Intel TDX module

Secure arbitration mode (SEAM) is an extension to the virtual machines extension (VMX) architecture to define a new VMX root operation called SEAM VMX root and a new VMX non-root operation called SEAM VMX non-root. Collectively, the SEAM VMX root and SEAM VMX non-root execution modes are called operations in SEAM. SEAM VMX root operation is designed to host a CPU-attested, software module called the Intel® Trust-Domain Extensions (Intel® TDX) module to manage virtual machine (VM) guests called Trust Domains (TD). Currently, the Intel TDX Module is the only SEAM module that the Intel P-SEAMLDR installs [1].

By default, BIOS loads the built-in version of TDX Loader and TDX module from the IFWI during the server booting. For development or upgrading purpose without re-flashing the BIOS, a debug or new version SEAMLDR and TDX module could be placed into the ESP partition. The BIOS loads the new or debug version from ESP on the next boot.

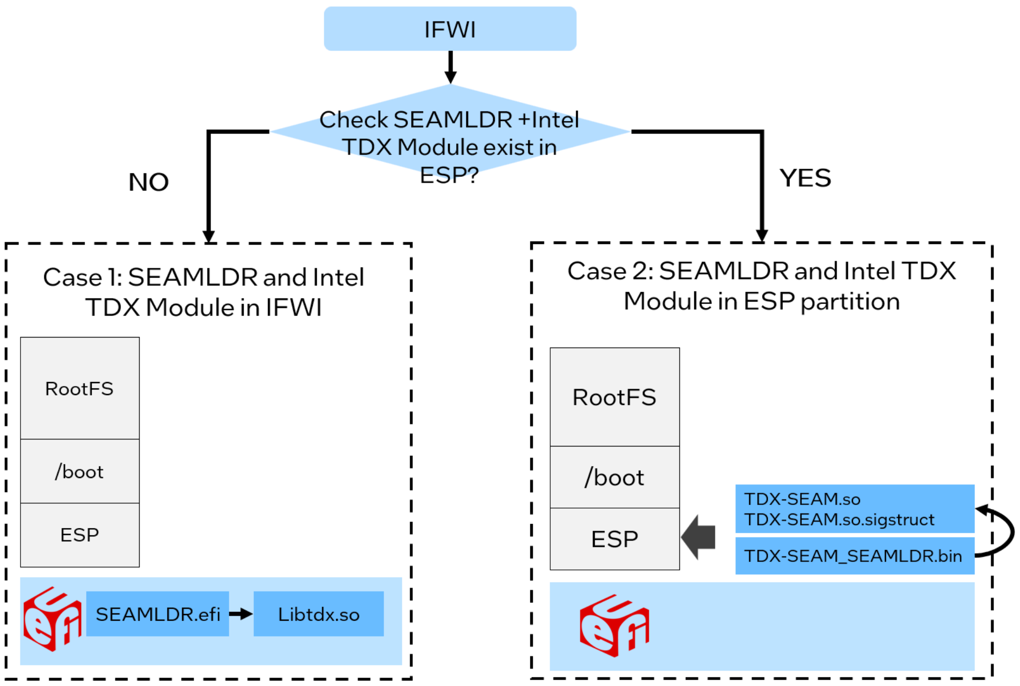

IFWI
Check SEAMLDR +Intel TDX Module exist in ESP?
NO
YES
Case 1: SEAMLDR and Intel TDX Module in IFWI
RootFS
/boot
ESP
SEAMLDR.efi
Libtdx.so
Case 2: SEAMLDR and Intel TDX Module in ESP partition
RootFS
/boot
ESP
TDX-SEAM.so
TDX-SEAM.so.sigstruct
TDX-SEAM_SEAMLDR.bin

Figure 32: BIOS Search TDX Module from ESP

The naming rule is:

- <ESP>/EFI/TDX/TDX-SEAM_SEAMLDR.bin
- <ESP>/EFI/TDX/TDX-SEAM.so
- <ESP>/EFI/TDX/TDX-SEAM.so.sigstruct

Check the updated TDX module information.

```
# It will display a string including version information of tdx module, such as
major version, monir version, build date, etc.

$ sudo cat /sys/firmware/tdx/tdx_module/*
0x00000000202302060x000001c90x000000010x00000000initialized0x00008086
```

6.2 Off-TD Debug via GDB from the Host

QEMU supports working with gdb via gdb's remote-connection facility (the "gdbstub"). This allows you to debug guest code in the same way that you might do with a low-level debug facility like JTAG on real hardware. You can stop and start the virtual machine, examine states like registers and memory, and set breakpoints and watchpoints. Refer to https://www.QEMU.org/docs/master/system/gdb.html for detailed gdb usage.

To support gdb the Intel TDX module exposes the following APIs:

- TDH.VP.RD/WR to allow QEMU emulator to read/write guest's CPU states.
- TDH.MEM.RD/WR to allow QEMU emulator to read/write guest memory.

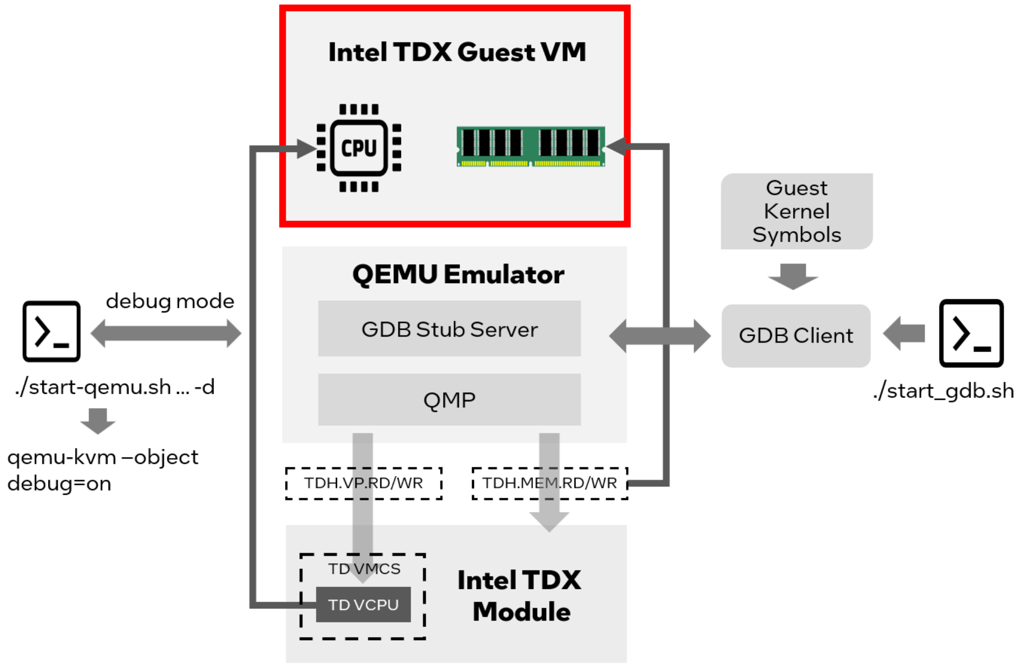


Figure 33: Off-TD Debug via GDB

Steps to debug TD guest are as follows:

- Step 1: Start TD guest in debug mode
  - Append "debug=on" to "-object". For example:

```
-object tdx-guest,id=tdx,debug=on
```

  - Add -s -S parameter to QEMU-kvm. For example:

```
$ QEMU-kvm -s -S
```

  - Disable kernel address randomization by append "nokaslr"

   - If booting TD via start-qemu.sh, please refer to below command to set “debug=on”:

```
$ ./start-qemu.sh -i <guest image> -k <guest kernel> -d
```

- Step 2: Install the guest kernel’s debug symbol into the host.

```
$ sudo dnf install intel-mvp-tdx-guest-kernel-debuginfo
```

- Step 3: Run the script start_gdb.sh with the following content.

```
#!/bin/bash
GDB=gdb
MOD_DIR=/usr/lib/debug/usr/lib/modules/<guest kernel>/
$GDB \
-ex "add-auto-load-safe-path $MOD_DIR" \
-ex "file $MOD_DIR/vmlinux" \
-ex "set arch i386:x86-64:intel" \
-ex "set remotetimeout 360" \
-ex "target remote 127.0.0.1:1234“
```

- Step 4: In the GDB console, use command “hb” to set the first break point.

```
gdb> hb start_kernel
```

The software breakpoint is available after the kernel is loaded into Guest Physical Address (GPA) space by QEMU.

6.3 Check Memory Encryption

There are lots of approaches to check whether TDX memory is encrypted or not. This section introduces how to do this check via a GDB debug approach.

1. Install the kernel development package on the host for debug symbols (using RHEL distro as example):

```
$ sudo dnf install intel-mvp-tdx-kernel-devel
```

2. Get the GVA (guest virtual address) of the `.text` code section of guest kernel.

```
$ # Extract the guest kernel binary
$ /usr/src/kernels/$(uname -r)/scripts/extract-vmlinux <path-to-guest-kernel-
file > vmlinux
$ objdump -d vmlinux > disassembled-vmlinux.asm && head -n 20 disassembled-
vmlinux.asm
...
ffffffff81000000 <.text>:
ffffffff81000000:    48 8d 25 51 3f c0 01     lea    0x1c03f51(%rip),%rsp
ffffffff81000007:    48 8d 3d f2 ff ff ff     lea    -0xe(%rip),%rdi
ffffffff8100000e:    56                       push %rsi
ffffffff8100000f:    e8 dc 06 00 00           callq 0xffffffff810006f0
...
```

The result shows that the virtual address of `.text` section starts from `0xffffffff81000000`.

3. Verify the instructions/memory at the guest physical address of the .text code section in a non-confidential VM guest.
   - ➢ Launch a non-confidential guest, `nokaslr` should be appended for kernel command like below to turn off the Kernel Address Space Layout Randomization (KASLR).

```
-append "root=/dev/vda1 console=hvc0 nokaslr"
```

   - ➢ Enter QEMU monitor shell

     If using start-qemu.sh, just "telnet 127.0.0.1 9001"

   - ➢ Disassemble the virtual address of the .text section

```
(QEMU) stop
(QEMU) x /10i 0xffffffff81000000
0x01000000:  48 8d 25 51 3f c0 01     leaq     0x1c03f51(%rip), %rsp
0x01000007:  48 8d 3d f2 ff ff ff     leaq     -0xe(%rip), %rdi
0x0100000e:  56                       pushq    %rsi
0x0100000f:  e8 dc 06 00 00           callq    0x10006f0
```

4. Verify the instructions/memory at guest physical address of .text code section in a TD guest.
   - Launch a TD guest
     - debug=on should be appended for QEMU command line

```
-object tdx-guest,id=tdx,debug=on
```

     - nokaslr should be appended for kernel command line

```
-append "root=/dev/vda1 console=hvc0 nokaslr"
```

   - Enter QEMU monitor shell

     If using start-QEMU.sh, just "telnet 127.0.0.1 9001"

   - Disassemble the virtual address of .text section

```
(QEMU) stop
(QEMU) x /10i 0xffffffff81000000
0xffffffff81000000:  98                          cwtl
0xffffffff81000001:  f8                          clc
0xffffffff81000002:  49 5e                       popq     %r14
0xffffffff81000004:  5a                          popq     %rdx
0xffffffff81000005:  55                          pushq    %rbp
...
```

The disassembled instructions should be different from a non-confidential guest and should look meaningless (all zero) since the memory is encrypted.

6.4 Run Intel AMX workload within TDX Guest

Intel AMX is a new built-in accelerator that improves the performance of deep-learning training and inference on the CPU. This is ideal for workloads like natural-language processing, recommendation systems, and image recognition.[12] It is available on the 4th Gen Intel® Xeon® Scalable processors. Use the following approach to check its capability on the host server and TD guest.

```
$ grep -o amx /proc/cpuinfo
```

Expect to see output of several "amx". Empty results mean Intel AMX is not enabled.

This section introduces how to run AI workload boosted by Intel AMX within an Intel TDX guest. This offers another layer of security over traditional VM's to protect the model data while it is being used.

- Install the Intel® Optimization for TensorFlow* version 2.8.0 via pip. Python versions supported are 3.7, 3.8, 3.9, 3.10. For TensorFlow versions 1.13, 1.14

[12] Intel® Advanced Matrix Extensions Overview

and 1.15 with pip > 20.0, if you get an “invalid wheel error”, try to downgrade the pip version to < 20.0

```
$ dnf install python3
$ python -m pip3 install intel-tensorflow-avx512==2.8.0
```

- Download a pre-trained model.

```
$ wget https://storage.googleapis.com/intel-optimized-tensorflow/models/v1_8/
mobilenet_v1_1.0_224_frozen.pb
```

- Clone the intelai/models repo and then navigate to the benchmark directory.

```
$ dnf install git
$ git clone https://github.com/IntelAI/models.git cd models/benchmarks
```

- Intel® Optimization for TensorFlow uses Intel® oneAPI Deep Neural Network Library (oneDNN) and OpenMP library. The DNNL_MAX_CPU_ISA environment variable can be used to limit processor features for oneDNN, it should be set to AVX512_CORE_AMX to use AMX features. The OMP_NUM_THREADS and KMP_AFFINITY environment variables set the number of threads and thread affinity for OpenMP library. Set these environment variables.

```
$ export DNNL_MAX_CPU_ISA=AVX512_CORE_AMX
$ export OMP_NUM_THREADS=16
$ export KMP_AFFINITY=granularity=fine,verbose,compact
```

- Run online inference. Replace <PATH> to the absolute path where the pre-trained model is located.

```
$ python3 launch_benchmark.py \
--benchmark-only --framework tensorflow --model-name mobilenet_v1 \
--mode inference --precision bfloat16 --batch-size 1 \
--in-graph /opt/mobilenet_v1_1.0_224_frozen.pb \
--num-intra-threads 16 --num-inter-threads 1 --verbose --\ input_height=224
input_width=224 warmup_steps=20 steps=20 \ input_layer='input'
output_layer='MobilenetV1/Predictions/Reshape_1'
```

- The result will look like the following.

```
[Running warmup steps...]
steps = 10, 360.33539518900346 images/sec steps = 20, 349.292471685543 images/sec
[Running benchmark steps...]
steps = 10, 364.1521097412745 images/sec steps = 20, 369.8028566390407 images/sec
Average Throughput: 364.37 images/s on 20 iterations
```

If running same workload without “**export DNNL_MAX_CPU_ISA=AVX512_CORE_AMX**”, the result will be a noticeably smaller (images/sec) because AMX is not being used to accelerate.

NOTE: If you fail to run above commands and see a message like "If you cannot immediately regenerate your protos, some other possible workarounds are: 1. Downgrade the protobuf package to 3.20.x or lower. 2. Set PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python (but this will use pure-Python parsing and will be much slower)." you have two options. Option 1 is to upgrade the protobuf version to 3.20.0 as following:

```
$ pip3.8 install --upgrade protobuf==3.20.0
```

Option 2 is to set the environment variable as following

```
$ export PROTOCOL_BUFFERS_PYTHON_IMPLEMENTATION=python
```

Then re-run above online inference command.

7 Disclaimer

The released components of the Linux Reference Stack for Intel TDX: Virtual Firmware (edk2/TDVF), bootloader (grub2), and the Linux kernel, are fully enabled to be run from within the Linux-based Intel TDX Guest VM to take advantage of the Intel TDX security technology for cryptographically isolating Trusted VMs from the rest of the system.

While Intel TDX removes the need for a Guest VM to trust the host and virtual machine manager (VMM), it cannot by itself protect the guest VM from host/VMM attacks that leverage existing paravirt-based communication interfaces between the host/VMM and the guest (such as MMIO, portIO, etc.). To achieve the full protection against such attacks, the Guest VM SW stack needs to be hardened to securely handle a untrusted and potentially malicious input from a host/VMM via the above-mentioned interfaces. This hardening effort is not specific to Intel TDX as a technology, but common for all confidential cloud computing solutions and the components of the VM guest SW stack. It should be an industry-wide effort together with the open-source maintainers to perform the security analysis and hardening of these components for the confidential computing threat model.

The Linux Reference Stack for Intel TDX team has invested a significant effort in hardening the Linux kernel that is released as part of the Linux Reference Stack for Intel TDX, the threat model for the Linux guest kernel, as well as the implemented mitigation mechanisms are explained in the Intel TDE Linux guest kernel security specification. The overall hardening methodology, as well as documentation on the tools that have been used can be found in Intel TDE guest Linux kernel hardening strategy. As a result, the Linux Reference Stack for Intel TDX kernel tree contains numerous patches that either implement these hardening mechanisms or fix the security issues that were discovered during the hardening process. It is strongly recommended that all these patches are manually carried forward to the intended production kernels, until they are merged into the mainline Linux kernel and will become part of the upstream base kernel tree. In particular, the following two patches that are critical for the security of the Intel TDX Linux guest kernel must be included in any production guest kernel:

Commit ID:

- c942fc241d4e6c215731b6f03740b1a8bfc42018 from patches-tdx-kernelMVP-KERNEL-5.19-v2.4.tar.gz

- Commit ID: c289330c56c61508a1008d74fc65b7bc24a4a7d5 from patches-tdx-kernelMVP-KERNEL-5.19-v2.4.tar.gz

It is important to note that the hardening of the Linux guest kernel has not been finalized for this release and other components, such as virtual firmware (edk2/TDVF) and the bootloader (grub2), still need more attention. In particular, the existing interfaces that edk2/TDVF or grub2 expose towards the host/VMM have not yet been analyzed for potential security implications against the confidential cloud computing threat model. It is strongly recommended that this analysis be done, and any issues uncovered are mitigated before these components are used in production.

8 References

[1] Intel, "Intel® TDX White Papers," February 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.

[2] Intel, "TDX Guest Hardening," [Online]. Available: https://intel.github.io/ccc-linux-guest-hardening-docs/tdx-guest-hardening.html.

[3] Confidential Computing Consortium, "A Technical Analysis of Confidential Computing," 2022.

[4] Trust Computing Group, TCG Guidance on Integrity Measurements and Event Log, 2021.