How to setup TD Partitioning environment

Table of Contents

1. Introduction

1.1 hardware prerequisite

- Platform: tested on Eagle Stream
- Baseboard: Archer City (So far haven't been successful with Quantas baseboard)
- SPR Step E0+ (Tested on: Sapphire Rapids XCC E2 stepping, 56 cores per socket, 1.8GHz/350W)
- IFWI: EGSDCRB.SYS.OR.64.2022.37.3.03.1114.1_EMR_EBG_SPS_IPClean.bin (Tested with this version).
- IFWI configuration: HW & BIOS (Pls refer to this page even to download the latest IFWI)
- TDX Module: https://ubit-artifactory-il.intel.com/artifactory/tdx_seam-il-local/TDX-SEAM/Release/Label/TDX_1.5.00.16.456/TDX_1.5.00.16.456.zip (Tested with this version)

```
tdx: TDX module: vendor_id 0x8086, major_version 1, minor_version 5, build_date 20230130, build_num 456
```

1.2. Prerequisite environment setup

The Wiki doesn't cover the instructions of how to setup nested virtualization environment and it assumes that users know how to create a guest Linux rootfs or create a Windows guest image, etc.

For users who are not super familiar with these, it's advised to look for other tutorials and setup a non-TDX Linux nested virtualization environment first. For simplicity, you can simply use the vanilla repos for all the components: Linux kernel, QEMU or virtual BIOS, and replace them with TDX enlightned versions later.

- Install a vanilla or TDX enligntened Linux on host (L0).
  - We only used Ubuntu in this project, but presumably other Linux distributions could be working as well.
- From L0 host, launch a Linux image (referred as L1_rootfs.img in this wiki) as L1 guest using QEMU.
  - L1_rootfs.img is launched as a virtio-blk device
  - Preferable networking is enabled on L1
- From L1 guest, launch a Linux image (referred as L2_rootfs.img in this wiki) as L2 guest using QEMU.
  - L2_rootfs.img is launched as a virtio-blk device

After the non-TDX nested environemnt is setup, you can enable TDX/TD Partitioning step by step according to this wiki.

Note:

- As of the time of writing, we used Ubuntu 20.04 cloud image as the base L1 image.

1.3 Repos and validated commit IDs

By default TD Partitioning is able to launch un-modified L2 Linux or Windows guests. To achieve best performance, we proposed the Virtual Passthrough architect to run enlightened L2 guest, which is coverred in section 5.

	Description	URL	validated commit id
L0 kernel	Running on host	https://github.com/intel/td-partitioning/tree/td_part_l0_vmm	b40f2038
L1 kernel	Kernel for L1 guest (TD)	https://github.com/intel/td-partitioning/tree/td_part_l1_vmm	9ae1a96d
L2 OS	L2 guest (TD Partitioning)	Windows 10, or Ubuntu distribution with vanilla Linux kernel	n/a
L1 QEMU	Used to launch L1 guest (TD) from TDX host	https://github.com/intel/qemu-td-partitioning/tree/td_part_l1_qemu	0cb15098
L2 QEMU	Used to launch L2 guest (TD) from L1 guest	https://github.com/intel/qemu-td-partitioning/tree/td_part_l2_qemu	f2e88f1a
L1 OVMF	Virtual BIOS for L1 guest	https://github.com/intel/ovmf-td-partitioning/tree/td_part_l1_ovmf	8850b77c
L2 OVMF	Virtual BIOS for L2 guest	https://github.com/intel/ovmf-td-partitioning/tree/td_part_l2_ovmf	0f0ec01c

2. Setup host (L0)

On a system that supports TD Partitioning.

```
$ git clone https://github.com/intel/td-partitioning.git
$ cd td-partitioning
$ git checkout td_part_l0_vmm
$ cp /boot/config-$(uname -r) .config
$ make olddefconfig
$ scripts/config --enable CONFIG_INTEL_TDX_HOST
$ scripts/config --enable CONFIG_KVM
$ scripts/config --enable CONFIG_KVM_INTEL
$ make && sudo make modules_install && sudo make install
```

Then make and install the kernel on the host. Edit the grub.cfg to add the following boot parameters to the host kernel: **numa_balancing=disable split_lock_detect=off**

If the kernel is built and installed properly, it's supposed to see the following from L0 kernel dmesg.

```
[    9.318048] tdx: TDX module initialized.
[    9.318052] kvm_intel: tdx: td partitioning supported
[    9.318058] kvm_intel: tdx: max servtds supported per user TD is 1
[    9.318071] kvm_intel: tdx: live migration supported
[    9.318073] kvm_intel: TDX is supported.
```

Notes:

- split_lock_detect=off: As the time of writing, the L2 OVMF boot code may access data across cache line boundaries which may trigger #AC. Adding this option is to prevent #AC from being injected to the L2 guest.

3. Setup L1 Guest (TD)

We did all the following setup on the host machine.

3.1 Build L1 QEMU

```
$ sudo apt install libspice-protocol-dev libspice-server-dev libslirp-dev
$ git clone https://github.com/intel/qemu-td-partitioning
$ cd qemu-td-partitioning
$ git checkout td_part_l1_qemu
$ git submodule update --init --recursive
$ mkdir build && cd build
$ ../configure --enable-kvm --target-list=x86_64-softmmu --enable-debug --enable-spice --enable-slirp
$ make
```

Notes:

- *--enable-spice* is needed to launch a L2 Windows guest, but optional for L2 Linux guests.
- *--enable-slirp* is needed for user type netdev device which is used in our sample guest launch scripts, and it's optional if you would like to setup guest network device in different ways. However, the following is the sample scripts to manually build and install libslirp when *--enable-slirp* is used.

```
$ sudo apt-get install git libglib2.0-dev libfdt-dev libpixman-1-dev zlib1g-dev ninja-build meson
$ git clone https://gitlab.freedesktop.org/slirp/libslirp.git
$ cd libslirp
$ meson build
$ ninja -C build install
```

3.2 Build L1 OVMF

```
$ git clone https://github.com/intel/ovmf-td-partitioning.git
$ cd ovmf-td-partitioning
$ git checkout td_part_l1_ovmf
$ git submodule update --init --recursive
$ source edksetup.sh
$ make -C BaseTools
$ OvmfPkg/build.sh -p OvmfPkg/IntelTdx/IntelTdxX64.dsc -a X64 -t GCC5 -DFD_SIZE_2MB -
DDEBUG_ON_SERIAL_PORT=TRUE
```

Note: The above -DDEBUG_ON_SERIAL_PORT=TRUE option is to determine whether to enable details OVMF logs in boot time. In order to be able to see the logs through serial port, need to add *-serial stdio* option to the QEMU launch script as well.

3.3 Build L2 OVMF

```
$ git clone https://github.com/intel/ovmf-td-partitioning.git
$ cd ovmf-td-partitioning
$ git checkout td_part_l2_ovmf
$ git submodule update --init --recursive
$ source edksetup.sh
$ make -C BaseTools
$ OvmfPkg/build.sh -p OvmfPkg/OvmfPkgX64.dsc -a X64 -t GCC5 -DFD_SIZE_2MB
```

3.4 Build L1 Kernel

```
$ git clone https://github.com/intel/td-partitioning.git
$ cd td-partitioning
$ git checkout td_part_l1_vmm
$ cp /boot/config-$(uname -r) .config
$ scripts/config --enable CONFIG_INTEL_TDX_GUEST
$ scripts/config --enable CONFIG_INTEL_TD_PART_GUEST
$ scripts/config --enable CONFIG_KVM
$ scripts/config --enable CONFIG_KVM_INTEL
$ scripts/config --enable CONFIG_VIRTIO_BLK
$ scripts/config --enable CONFIG_VIRTIO_IOMMU
$ make
```

Attached L1_VMM_CONFIG is a complete config file for L1 kernel. Please note that you need to copy the modules to L1 rootfs. Sample (but not complete) scripts:

```
$ cd td-partitioning
$ make modules_install INSTALL_MOD_PATH=${local_path}
$ scp ${local_path} to L1 rootfs:/lib/modules/
```

3.5 Sample scripts to launch a L1 Linux guest (TD)

Notes:

- L2 OVMF is loaded to memory when we are launching the L1 guest and the L2 QEMU launch script doesn't need to include L2 OVMF binary.
- This is a sample only, and not all QEMU arguments and Linux kernel boot parameters are mandatary.

```
#!/bin/bash

MEMORY=8G
CPU=20

# L1 QEMU: td_part_l1_qemu
QEMU=/path/to/qemu-system-x86_64

# L1 Kernel: td_part_l1_vmm
KERNEL=/path/to/bzImage

# L1 BIOS: td_part_l1_ovmf
BIOS=/path/to/Build/IntelTdx/DEBUG_GCC5/FV/OVMF.fd

# L2 BIOS: td_part_l2_ovmf
L2BIOS=/path/to/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd

sudo $QEMU -m $MEMORY \
        -nographic \
        -serial null \
        -vga none \
        -smp ${CPU} \
        -cpu host,host-phys-bits,pmu=off,pks=on \
        -device virtio-serial,romfile= \
        -chardev stdio,id=virtiocon0,mux=on,signal=off,logfile=./logs/vm_log_$(date +"%FT%H%M").log \
        -device virtconsole,chardev=virtiocon0 \
        -monitor chardev:virtiocon0 \
        -netdev user,id=unet,hostfwd=tcp::2223-:22,hostfwd=tcp::5900-:5900 -device virtio-net-pci,netdev=unet \
        -bios ${BIOS} \
        -kernel $KERNEL \
        -initrd /path/to/initrd \
        -drive if=virtio,format=raw,media=disk,index=0,file=L1_rootfs.img \
        -machine q35,accel=kvm,l2bios=${L2BIOS} \
        -object memory-backend-memfd-private,id=ram1,size=$MEMORY \
        -object tdx-guest,id=tdx0,debug=on,sept-ve-disable=on,num-l2-vms=3 \
        -machine kernel-irqchip=split,sata=off,pic=off,pit=off,memory-backend=ram1 \
        -machine confidential-guest-support=tdx0 \
        -append "root=/dev/vda1 rw console=hvc0 nomce no-kvmclock no-steal-acc ignore_loglevel nopat
memmap=1023M\$1M memmap=2G\$4G"
```

If the L1 Linux guest launches properly, dmesg should indicate that it boots as a TD.

```
[    0.000000] tdx: Guest detected
[    0.000000] TDX: Enabled TDX guest device filter
[   13.156016] Memory Encryption Features active: Intel TDX
```

4. Enable L2 guest (TD Partitioning)

All the following are performed on the L1 guest. With the above L1 launch script, you can remote login to L1 guest from host:

```
$ ssh -X -p 2223 sdp@localhost
```

4.1 Build L2 QEMU

On L1 guest, download and build L2 QEMU similar to 3.1 Build L1 QEMU. Please note that it's on td_part_l2_qemu branch to build L2 QEMU.

4.2 Launch L2 Linux

On L1 guest, create a L2 rootfs, fo example download Ubuntu 20.04 cloud image or Ubuntu 22.04 cloud image, and rename it to L2_rootfs.img.

Notes: This is a sample only, and not all QEMU arguments or Linux kernel boot parameters are mandatary.

```
#!/bin/bash

MEMORY=3G

# td_part_l2_qemu
QEMU=/path/to/qemu-system-x86_64

KERNEL=/path/to/vanilla_linux_kernel
INITRD=/path/to/vanilla_initrd

sudo $QEMU -m $MEMORY \
        -nographic \
        -enable-kvm \
        -serial stdio \
        -cpu host,-mtrr \
        -smp 4,sockets=1,maxcpus=4 \
        -nodefaults \
        -kernel $KERNEL \
        -initrd /$INITRD \
        -append "root=/dev/vda1 ro console=ttyS0 ignore_loglevel earlyprintk=ttyS0 "  \
        -drive if=virtio,format=raw,file=L2_rootfs.img \
        -netdev user,id=unet,hostfwd=tcp::2222-:22 -device virtio-net-pci,netdev=unet \
        -object memory-backend-file,mem-path=/dev/mem,size=${MEMORY},share=on,id=mem0 \
        -M q35,accel=kvm,kvm-type=td-part,memory-backend=mem0,max-ram-below-4g=1G
```

Currently TD Partitioning doesn't support vCPU migration, thus by default the L2 vCPU is affinitized to L1 vCPU in the 1:1 fasion. If you want different vCPU affinity, the optional QEMU option *-vcpu* is available. For example:

```
        -vcpu vcpunum=0,affinity=2 \
        -vcpu vcpunum=1,affinity=3 \
        -vcpu vcpunum=2,affinity=4 \
        -vcpu vcpunum=3,affinity=5 \
```

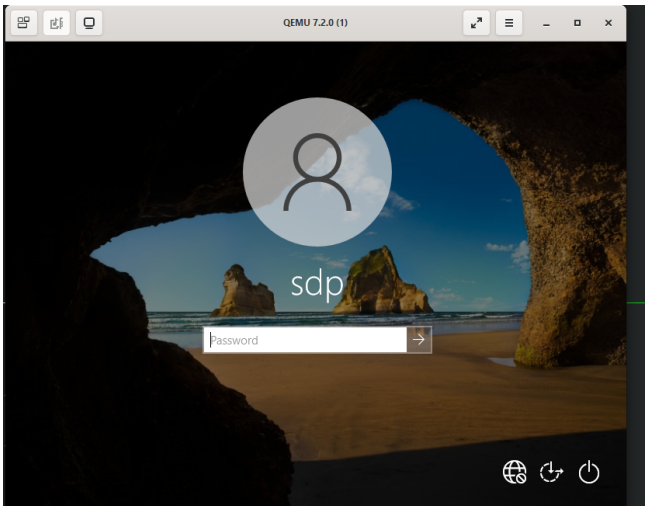
4.4 Launch L2 Windows

In L1 guest, create a Windows image (referred as win10-uefi.img in following script). How to prepare such an image is not covered by this document.

```
#!/bin/sh

MEMORY=3G

# td_part_l2_qemu
QEMU=/path/to/qemu-system-x86_64

sudo $QEMU -enable-kvm \
        -cpu host \
        -smp 8,sockets=1 \
        -m ${MEMORY} \
        -vga none \
        -device ramfb \
        -serial stdio \
        -monitor telnet:127.0.0.1:1698,server,nowait \
        -netdev user,id=unet,hostfwd=tcp::2222-:22 -device virtio-net-pci,netdev=unet \
        -object memory-backend-file,mem-path=/dev/mem,size=${MEMORY},share=on,id=mem0 \
        -M q35,accel=kvm,kvm-type=td-part,memory-backend=mem0,max-ram-below-4g=1G \
        -spice port=5900,disable-ticketing=on \
        -drive file=win10-uefi.img,if=virtio
```

You can access to the Windows GUI from your Windows host. Download and install virt-view, then open a Windows *Command* shell and run the following command, assuming 10.23.0.114 is the IP address of your L0 host.

```
C:\> "c:\Program Files\VirtViewer v11.0-256\bin\remote-viewer.exe" spice://10.23.0.114:5900
```



5. Setup TD Partitioning with Virtual Passthrough support

5.1 Repos and validated commit IDs

To achieve best I/O performance on L2 guests, we enligntened L2 guest kernel to virtually passthrough I/O device from host to L2 guest.

	Description	URL	validated commit id
L2 kernel	L2 kernelKernel for L2 guest (TD Partitioning)	https://github.com/intel/td-partitioning/tree/td_part_l2_vpt_stable-6.2.16	2f5fe0fd

5.2 Launch a L1 Linux guest (TD)

5.2.1 Prepare bridge virbr0

Here is an example with libvirt. it's possible to use other methods.

```
# Install libvirt packages to create virbr0
$ apt install libvirt-daemon libvirt-daemon-system bridge-utils
$ systemctl start libvirtd
$ brctl show
bridge name     bridge id               STP enabled     interfaces
virbr0          8000.525400c08d38       yes
$ ifconfig
```

5.2.2 Sample scripts to launch a L1 Linux guest (TD)

Compared with the scripts listed in section 3.5, the major difference is to create two virtio-blk devices and two virtio-net devices: one pair is used the L1 guest and the other pair of virtio devices will virtual passthrough to L2 guest later.

```
#!/bin/bash

SMP=8
MEM=12G
QUEUES=$(echo "${SMP} / 2"|bc)

# L1 QEMU: td_part_l1_qemu
QEMU=/path/to/qemu-system-x86_64

# L1 Kernel: td_part_l1_vmm
KERNEL=/path/to/bzImage

# L1 BIOS: td_part_l1_ovmf
BIOS=/path/to/Build/IntelTdx/DEBUG_GCC5/FV/OVMF.fd
# L2 BIOS: td_part_l2_ovmf
L2BIOS=/path/to/Build/OvmfX64/DEBUG_GCC5/FV/OVMF.fd

# L1 rootfs
L1_ROOT_DISK=/path/to/L1_rootfs.img
# L2 rootfs
L2_ROOT_DISK=/path/to/L2_rootfs.img

APPEND="root=/dev/vda1 rw console=hvc0 nomce no-kvmclock no-steal-acc ignore_loglevel nopat memmap=1023M\$1M
memmap=3G\$4G earlyprintk=ttyS0"

sudo $QEMU \
        -name tdp,debug-threads=on \
        -smp ${SMP},sockets=1 \
        -cpu host,host-phys-bits,pmu=off,pks=on,-monitor \
        -no-hpet -nographic -vga none \
        -nodefaults \
        -object memory-backend-memfd-private,id=ram1,size=${MEM} \
        -object tdx-guest,id=tdx0,debug=on,sept-ve-disable=on,num-l2-vms=2 \
        -machine kernel-irqchip=split,sata=off,pic=off,pit=off,confidential-guest-support=tdx0,memory-
backend=ram1 \
        -machine q35,accel=kvm,l2bios=${L2BIOS} \
        -bios ${BIOS} \
        -kernel ${KERNEL} \
        -append "${APPEND}" \
        -drive if=none,cache=none,file=${L1_ROOT_DISK},id=drive0 -device virtio-blk-pci,drive=drive0,
iommu_platform=true,disable-legacy=on \
        -netdev tap,id=tap0,queues=${QUEUES},script=./qemu-ifup,downscript=no -device virtio-net-pci,mq=true,
netdev=tap0,mac=52:54:00:22:34:50,iommu_platform=true,disable-legacy=on \
        -drive if=none,cache=none,file=${L2_ROOT_DISK},id=drive1 -device virtio-blk-pci,drive=drive1,
iommu_platform=true,disable-legacy=on \
        -netdev tap,id=tap1,queues=${QUEUES},script=./qemu-ifup,downscript=no -device virtio-net-pci,mq=true,
netdev=tap1,mac=52:54:00:23:35:50,iommu_platform=true,disable-legacy=on \
        -monitor telnet:127.0.0.1:1235,server,nowait \
        -chardev stdio,id=mux,mux=on \
        -device virtio-serial,romfile= \
        -device virtconsole,chardev=mux -monitor chardev:mux \
        -serial chardev:mux \
```

Note: put qemu-ifup in the host's execution path.

If the L1 Linux guest launches properly, *lspci* should be able to see two virtio-blk devices and two virtio-net devices.

```
$ lspci |grep Virtio
00:01.0 SCSI storage controller: Red Hat, Inc. Virtio block device (rev 01)
00:02.0 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
00:03.0 SCSI storage controller: Red Hat, Inc. Virtio block device (rev 01)
00:04.0 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
```

5.3 Prepare for Virtual Passthrough enabled L2 guest

All the following are performed on the L1 guest.

5.3.1 Build enlightened L2 kernel

It can be built similar to Build L1 Kernel, but on a different branch *td_part_l2_vpt*. It's supposed that you can reuse the L1 kernel .config directly.

```
$ git clone https://github.com/intel/td-partitioning.git
$ cd td-partitioning
$ git checkout td_part_l2_vpt
$ cp /boot/config-$(uname -r) .config
$ scripts/config --enable CONFIG_INTEL_TDX_GUEST
$ scripts/config --enable CONFIG_INTEL_TD_PART_GUEST
$ scripts/config --enable CONFIG_KVM
$ scripts/config --enable CONFIG_KVM_INTEL
$ scripts/config --enable CONFIG_VIRTIO_BLK
$ scripts/config --enable CONFIG_VIRTIO_IOMMU
$ make
# copy modules to L2 rootfs
```

5.3.2 Launch L2 Linux

5.3.2.1 Prepare virtual passthrough devices

Bind one virtio-blk device and one virtio-net device to VFIO driver.

```
$ lspci |grep Virtio
00:01.0 SCSI storage controller: Red Hat, Inc. Virtio block device (rev 01)
00:02.0 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)
00:03.0 SCSI storage controller: Red Hat, Inc. Virtio block device (rev 01)
00:04.0 Ethernet controller: Red Hat, Inc. Virtio network device (rev 01)

$ lspci -s 00:03.0 -n
00:03.0 0100: 1af4:1042 (rev 01)

$ lspci -s 00:04.0 -n
00:04.0 0200: 1af4:1041 (rev 01)

$ modprobe vfio-pci
$ echo 1 > /sys/module/vfio/parameters/enable_unsafe_noiommu_mode

$ echo 0000:00:03.0 > /sys/bus/pci/devices/0000\:00\:03.0/driver/unbind
$ echo 1af4 1042 > /sys/bus/pci/drivers/vfio-pci/new_id
$ echo 0000:00:04.0 > /sys/bus/pci/devices/0000\:00\:04.0/driver/unbind
$ echo 1af4 1041 > /sys/bus/pci/drivers/vfio-pci/new_id

$ modprobe kvm_intel ple_gap=0
$ echo 1 > /sys/module/vfio_iommu_type1/parameters/allow_unsafe_interrupts
```

5.3.2.2 Sample scripts to launch L2 Linux with virtual passthrough:

```
#!/bin/bash

SMP=8
MEM=4G

# td_part_l2_qemu
QEMU=/path/to/qemu-system-x86_64

# L2 Kernel: td_part_l2_vmm
KERNEL=/path/to/bzimage

APPEND="root=/dev/vda1 rw console=tty0 console=ttyS0 ignore_loglevel nopat nokaslr earlyprintk=ttyS0"

sudo $QEMU \
-smp ${SMP},sockets=1 \
-no-hpet -nographic -vga none \
-nodefaults \
-kernel ${KERNEL} \
-append "${APPEND}" \
-monitor telnet:127.0.0.1:1234,server,nowait \
-object memory-backend-file,mem-path=/dev/mem,size=${MEM},share=on,id=mem0 \
-device vfio-pci,host=00:03.0,x-no-mmap=true \
-device vfio-pci,host=00:04.0,x-no-mmap=true \
-serial stdio \
-cpu host,host-phys-bits \
-M q35,accel=kvm,sata=off,kvm-type=td-part-enlighten,vfio-identity-bars=true,vfio-allow-noiommu=true,memory-
backend=mem0,max-ram-below-4g=1G
```

Notes:

- *-device vfio-pci,host=<b:d.f>*, here <b:d.f> should comply with the real *lspci* result.
- To launch the enlightened L2 guest, we use *kvm-type=td-part-enlighten*, other than *kvm-type=td-part* in the previous session.
- The ddtionaly *-M vfio-identity-bars=true* to make it possible to bypass virtio-iommu.
- This is a sample only, and not all following QEMU arguments and Linux kernel boot parameters are mandatary.

5.4 Virtio passthrough with virtio-IOMMU

To achieve optimal I/O performance, in the above configuration, we take advantage the fact that "TDP GPA == L1 VMM GPA" in TD Partition setup, to disable virtio-IOMMU and skip such GPA translation. If virtio-IOMMU is still needed, we can bring back virtio-IOMMU with following changes, though this may have some impacts on certain workloads like iperf3.

5.4.1 Launch L1 VMM with virtio-IOMMU

Comparing with the QEMU scripts of launching L1 guest in section 5.2.2, there are two modifications are needed:

1. Add the line "*-device virtio-iommu-pci*" to QEMU command line to add virtio-IOMMU.
2. Add "*iommu.passthrough=1 tdx_allow_acpi=VIOT*" to the L1 kernel boot parameters list ${APPEND}.

5.4.2 Launch L2 TDP guest with virtio-IOMMU

The L2 guest is also launched with some differences compared with section 5.4.2.

- Don't need to write 1 to /sys/module/vfio/parameters/enable_unsafe_noiommu_mode.
- Remove "*vfio-allow-noiommu=true"* from the QEMU launch script.