TRS Documentation

Linaro

Apr 19, 2023

ABOUT

The main documentation for the site is organized into a couple sections:

- *About*
- *Installation*
- *Security*
- *Features*

CHAPTER

ONE

ABOUT TRS

Developing software on its own is complicated and requires time, skills and lots of efforts. But being good at writing individual software isn't sufficient in this day and age. Systems are inherently complicated with lots of components interacting with each other. We have to deal with intra communication as well as external communication with remote systems. All aspects of security has to be considered, standards needs to be addressed and systems needs to be tested not only as individual components, but as coherent systems. For device manufacturers this becomes a real challenge, which is very costly both in terms on time and effort.

As an answer to the challenges presented, Linaro have created **TRS (Trusted Reference Stack)**, which is an umbrella project and a software stack containing well tested software components making up a solid base for efficient development and for building unique and differentiating end-to-end use cases.

1.1 Goals and key properties

- Common platform for deliverables from Linaro.
- Include all Linaro test suites and test frameworks making CI/CD and regression testing fast, valuable and efficient.
- Efficient development environment for engineers.
- A product ready reference implementation.
- Configurable to be able to meet different needs.
- Common ground and building block for Blueprints and similar targets.
- Interoperability making it possible to use alternative implementations.
- Pre-silicon IP support in environments like QEMU etc.

1.2 Firmware Software Components

The firmware components for TRS is provided by *Trusted Substrate* more details, please look *here*

1.3 Releases

1.3.1 v0.2 - 2023-03-07

- **Stable CI**
  - xtest from OP-TEE (nightly and merge request)
  - Measured boot tests (nightly and merge request)
  - Secure boot (nightly and merge request)
  - ACS 1.0 manually, except QEMU, where it is in CI.
- **Platform support, meaning that they work with TRS**
  - QEMU
  - RockPi4
  - Synquacer
- **New features**
  - Authenticated policies.
  - Grub as part of the boot flow.

1.3.2 v0.1 - 2022-12-16

- Restructured the layer structure, by moving some layers up to the top level.
- QEMU is built by Yocto instead of relying on the host installed QEMU version.
- Changed repo release/branching strategy.
- Trusted Substrate documentation has moved into a subsection of TRS.
- Uses Trusted Substrate v0.2.
- Uses LEDGE Secure v0.1.
- Features enabled: LUKS disc encryption, Measured Boot, UEFI Secure Boot using U-boot.

1.3.3 v0.1-beta - 2022-09-02

Note: This release is slightly flawed, mostly due to the fact that code was checked out when the build was started and the code did not always track stable commits.

- Builds TRS for the QEMU target.
- Boot cleanly up to the login prompt.
- Nothing tested.
- RockPi4 works, but not officially part of the v0.1-beta release.

CHAPTER

TWO

GETTING STARTED

The instructions on this page are a one time setup (per workspace). Two installation/setup methods are provided below. First is the manual option. This is for those who may want to integrate into their native development environment. The second option is to create create a development environment in docker. This will mean having Docker available on your development system.

CHAPTER

THREE

MANUAL INSTALLATION

3.1 1. Install repo

Note that here you don’t install a huge SDK, it’s simply a Python script that you download and put in your `$PATH`, that’s it. Exactly how to “install” repo, can be found at the Google repo pages, so follow those instructions before continuing.

3.2 2. Getting the source code

Now we will check out code for the TRS. This step is light weight and only check out code necessary to build TRS. There are two flavors right now, either you checkout the one tracking lastest on all gits or you’ll checkout a certain release (the difference is in the `repo init` line, as highlighted).

3.2.1 For latest, do this

```
$ mkdir trs-workspace
$ cd trs-workspace
$ repo init -u https://gitlab.com/Linaro/trusted-reference-stack/trs-manifest.git -m default.xml
$ repo sync -j3
```

3.2.2 For a specific release, do this

```
$ mkdir trs-workspace
$ cd trs-workspace
$ repo init -u https://gitlab.com/Linaro/trusted-reference-stack/trs-manifest.git -m default.xml -b <release-tag>
$ repo sync -j3
```

3.3 3. Installing prerequisites

TRS depends on a couple of packages that needs to be present on the host system. These are installed as distro packages and using Python pip.

3.3.1 Host packages

Ubuntu / Debian

Host / apt packages

This will require your **sudo** password, from the root of the workspace:

```
$ cd <workspace root>
$ make apt-prereqs
```

This will install the following packages:

```
acpica-tools
adb
autoconf
automake
bc
bison
build-essential
ccache
chrpath
cloud-guest-utils
cpio
cscope
curl
device-tree-compiler
diffstat
expect
fastboot
file
flex
ftp-upload
gawk
gdisk
inetutils-ping
iproute2
libattr1-dev
libcap-dev
libfdt-dev
libftdi-dev
libglib2.0-dev
libgmp3-dev
libhidapi-dev
libmpc-dev
libncurses5-dev
libpixman-1-dev
libssl-dev
```

(continues on next page)

(continued from previous page)

```
libtool
locales-all
lz4
make
make
mtools
netcat-openbsd
ninja-build
pip
python3-cryptography
python3-pip
python3-pyelftools
python3-serial
python3-venv
python-is-python3
qemu-system-aarch64
rsync
sudo
unzip
uuid-dev
wget
xdg-utils
xdg-utils
xterm
xz-utils
zlib1g-dev
zstd
```

Arch Linux

> **Warning:** Just boiler plate, no complete instructions. Only Ubuntu versions tested so far.

Install the necessary packages using pacman.

```
$ sudo pacman -Syy
$ sudo pacman -S git
```

Fedora

> **Warning:** Just boiler plate, no complete instructions. Only Ubuntu versions tested so far.

Install the necessary packages using dnf.

```
$ sudo dnf update
$ sudo dnf install git
```

3.3.2 Python packages

By default all python packages will be installed at `<workspace root>/.pyvenv` using a virtual Python enviroment. The benefits by doing so is that if we delete the `.pyvenv` folder, there will be no traces left of the Python packages needed for TRS. It can eventually also avoid clashing with tools needing other versions of some Python packages.

```
$ cd <workspace root>
$ make python-prereqs
```

3.4 4. Building

3.4.1 4.1 Support virtualization with Xen (Optional)

To support Xen in TRS, in the configuration file `meta-trs/conf/distro/trs.conf` you need to replace distro feature `ewaol-baremetal` with `ewaol-virtualization` and append to variable `DISTRO_FEATURES`; with the virtualization feature, Xen hypervisor and its associated packages (including kernel modules and tools) will be built in TRS image.

```
# In the file meta-trs/conf/distro/trs.conf
DISTRO_FEATURES:append = " ewaol-virtualization"
```

3.4.2 4.2 Build firmwares and TRS image

Since we are using a virtual Python environment, we need begin by sourceing it.

```
$ source <workspace root>/.pyvenv/bin/activate
```

Note: The `source` command must be run **once** each time a new shell is created.

Next we start the build, this will probably take several hours on a normal desktop computer the first time you're building it with nothing in the cache(s). The TRS is based on various Yocto layers and if you don't have your `DL_DIR` and `SSTATE_DIR` set as an environment variable, those will be set to `$HOME/yocto_cache` by default. Note that the `clean` target does not remove the download and sstate caches. `make clean` is a rather quick process that is often needed after modifying the Yocto meta layers.

```
$ cd <workspace root>
$ make
```

After you complete the whole building process, if you want to only build firmwares for saving time, you could use the command `make meta-ts`; for only building the TRS image, the command `make trs` can be used. You also can use the command `make trs-dev`, it builds TRS image with enabling *ewaol-sdk* distro feature and includes debugging and profiling tools (e.g. gdb, perf, systemtap, ltt-ng, etc).

If you only want to build the firmare for a single target, you can choose the target from `meta-ts/meta-trustedsubstrate/conf/templates/multiconfig/` and run:

```
$ cd <workspace root>
$ make TS_SUPPORTED_TARGETS=<target-name> meta-ts
```

Only the firmware is target-specific. The image is shared across devices (note that not all targets that are supported by the firmware are supported by the TRS image).

3.5 5. Target specific installation

After following the steps above, please continue with the target specific instructions:

1. *Install QEMU*
2. *Run on bare-metal*

3.6 5. Tips and tricks

3.6.1 5.1 Reference local mirrors

As the repo forest grows, the amount of time to run the initial `repo sync` increases. The repository tool is able to reference a locally cloned forest and clone the bulk of the code from there, taking just the eventual delta between local mirrors and upstream trees. The way to do this is to add the parameter `--reference` when running the `repo init` command, for example:

```
$ repo init -u https://... --reference <path-to-my-existing-forest>
```

3.6.2 5.2 Local manifests

In some cases we might want to use another remote, pick a certain commit or even a add another repository to the current repo setup. The way to do that with repo is to use local manifests. The end result would be the same as manually clone or checkout a certain tag or commit. The advantage of using a local manifest is that when running "repo sync", the original manifest will not override our temporary modifications. I.e., it's possible to reference and keep using a temporary copy if needed.

CHAPTER

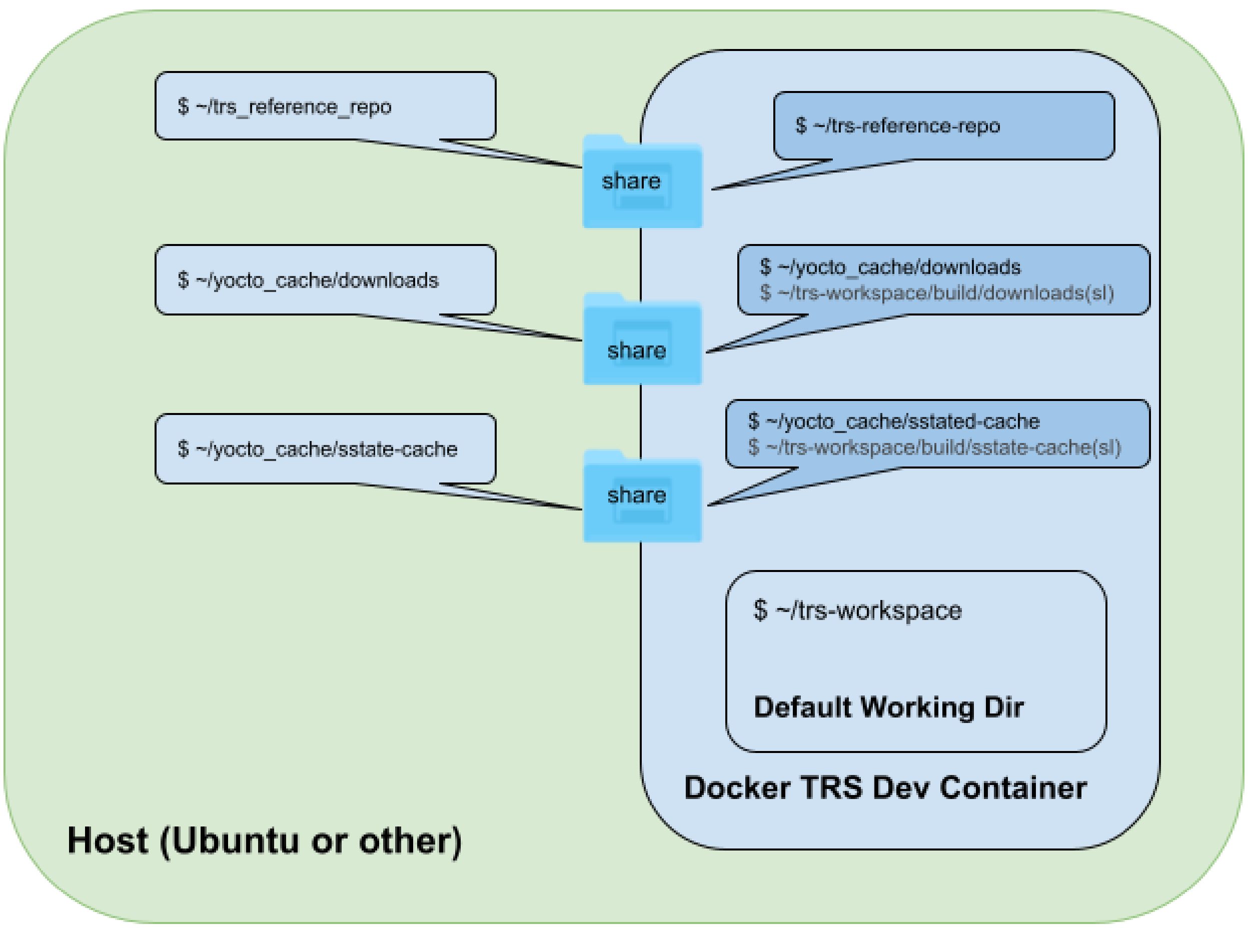
FOUR

DOCKER INSTALL

This installation method has been created to aid developers in quickly setting up an initial TRS development environment. By leveraging the scripts and Dockerfile available in the trs repository, with just a few steps you can have a trs-development environment running in a docker container. The benefits of using a container for your development environment include quickly reproducing your environment, speed of setup, all devs in a similar environment, can be customized/extended to meet your needs, usable across different host platforms, and more.

4.1 Container Configuration

This section provides an overview of how this container is set up.



Referring to the diagram above:

- The username is `dev`
- When logging into the container, it defaults into the pre-determined `$HOME/trs-workspace` directory
- Under `$HOME/trs-workspace` is the `./build` directory that has a softlink to the `$HOME/yocto_cache/` directories
- This docker configuration provides three shared directories
  - The first, `$HOME/trs_reference_repo` on the Host is shared with `$HOME/trs-reference-repo` in the container. This allows a user to keep it updated from the host side and potentially be shared by multiple containers
  - The second and third directories are tied to the creation of a yocto build cache, also to reduce build times. These default to `$HOME/yocto_cache` on the host and container. Two subdirectories are created under `$HOME/yocto_cache`. These are `$HOME/yocto_cache/sstate-cache` and `$HOME/yocto_cache/downloads`
- The default directories/shares described above may of course all be customized by modifying the Dockerfile and Scripts, but note that the naming must be assured to be consistent in all the files.

4.2 Tested Environments

The instructions/scripts in this section have been verified against Ubuntu 22.04 desktop machine and a share server environment also based on Ubuntu 20.04

4.3 Host Prerequisites

- Assure that Docker has been installed on your Host development machine

```
$: docker --version;
Docker version 20.10.19, build d85ef84;
```

Note: These instructions assume the user name is "dev"

4.4 Installation instructions

Since there are instructions for both the Host running Docker and the Container that will have the Ubuntu 20.04 TRS development environment set up, the following sections will delineate the difference by using "Host" or "Container" in the header. That way a user will know where the commands are intended to run.

4.4.1 1. Clone the TRS repository (Host)

Cloning the repo to be able to easily grab the scripts.

```
$ cd ~
$ mkdir trs-repo
$ cd trs-repo
$ git clone https://gitlab.com/Linaro/trusted-reference-stack/trs.git
```

Optionally check that the Dockerfile and scripts are present:

```
$ ls ~/trs-repo/trs/scripts/docker-scripts
Dockerfile  run-trs.sh  trs-install.sh
```

4.4.2 2. Build Docker Image (Host)

Create a docker image the named "trs"

```
$ cd ~/trs-repo/trs/scripts/docker-scripts
$ docker build -t trs .
```

Note: The above defaults to a UID/GID of 1000/1000; typical of an Ubuntu Desktop. If the host has a different UID/GID and it's desired for the container to have the same, use the following command instead of the one above:

```
$ cd ~/trs-repo/trs/scripts/docker-scripts
$ docker build --build-arg USER_UID=$(id -u) --build-arg USER_GID=$(id -g) -t trs .
```

Hint: During a docker build, it's not uncommon to see warnings such as the following that can be ignored.

For example

```
WARNING: apt does not have a stable CLI interface. Use with caution in scripts.
```

Optionally, after completion of the docker build, you can confirm that the images are there and look OK. Assuming you had no other docker images, you should see something similar to the following:

```
$ docker images
REPOSITORY   TAG       IMAGE ID       CREATED          SIZE
trs          latest    2a10a95eacd2   10 seconds ago   336MB
ubuntu       22.04     a8780b506fa4   4 weeks ago      77.8MB
```

4.4.3 3. Download and sync the TRS source using Repo tool (Host)

As described above, the Host and Container share the TRS repo in a shared directory. This section sets up this share TRS repo with the following commands.

```
$ cd ~
$ mkdir trs_reference_repo
$ cd trs_reference_repo
$ repo init -u https://gitlab.com/Linaro/trusted-reference-stack/trs-manifest.git -m␣
↪default.xml
$ repo sync
```

With all the above steps completed, we're now ready to launch the TRS container!

Warning: The location above is important as this is a shared folder between the Host and the Container. If the user chooses to change this location, the scripts/Dockerfile must be updated to align.

4.4.4 4. Create and enter the Container (Host)

The following commands will launch the container using the Dockerfile built in the earlier steps

```
$ cd ~/trs-repo/trs/scripts/docker-scripts
$ docker build -t trs .
$ ./run-trs.sh

dev@2d0b8419dac3:~/trs-workspace$
```

A new prompt will be shown in your terminal similar to the above and you're now working in the docker container!

Optionally, **from the Container**, some quick checks can be executed to assure that the container is set up right. This includes assuring all the shares have permissions set correctly, and that the build directory is linked to the yocto_cache directory using a soft link.

```
dev@92fae72fafee:~/trs-workspace$ ls -l
total 8
drwxr-xr-x 1 dev dev 4096 Jan 27 21:22 build
-rwxrwxr-x 1 dev dev 1936 Jan 27 20:41 trs-install.sh
dev@92fae72fafee:

dev@2d0b8419dac3:~/trs-workspace$ ls -l build
total 0
lrwxrwxrwx  1 dev dev    31 Jan 27 21:22 downloads -> /home/dev/yocto_cache/downloads
lrwxrwxrwx  1 dev dev    34 Jan 27 21:22 sstate-cache -> /home/dev/yocto_cache/sstate-
↪caches
dev@92fae72fafee:

dev@2d0b8419dac3:~/trs-workspace$ ls -l ~
total 16
drwxrwxr-x 17 dev  dev  4096 Jan 19 23:26 trs-reference-repo
drwxr-xr-x  1 dev  dev  4096 Jan 27 21:27 trs-workspace
drwxr-xr-x  1 root root 4096 Jan 27 21:21 yocto_cache
```

(continues on next page)

(continued from previous page)

```
dev@92fae72fafee:~/trs-workspace$ ls ~/yocto_cache -l
total 80
drwxrwxr-x   4 dev dev 73728 Jan 27 22:05 downloads
drwxrwxr-x 259 dev dev  4096 Jan 27 21:28 sstate-cache

dev@2d0b8419dac3:~/trs-workspace$

dev@92fae72fafee:~/trs-workspace$ ping google.com
PING google.com (142.250.188.238) 56(84) bytes of data.
64 bytes from lax31s15-in-f14.1e100.net (142.250.188.238): icmp_seq=1 ttl=116 time=31.7␣
↪ms
64 bytes from lax31s15-in-f14.1e100.net (142.250.188.238): icmp_seq=2 ttl=116 time=29.2␣
↪ms
64 bytes from lax31s15-in-f14.1e100.net (142.250.188.238): icmp_seq=3 ttl=116 time=26.4␣
↪ms
^C
--- google.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2001ms
rtt min/avg/max/mdev = 26.419/29.119/31.708/2.160 ms
dev@92fae72fafee:~/trs-workspace$ ^C
dev@92fae72fafee:~/trs-workspace$
```

If the user, group and shares all look good and a ping verified we have connectivity to the internet, then we're ready to move on to the final step, which is performing a trs build!

6 Build TRS (**Container**) ================================ To verify everything is correct, perform a build by executing the `./trs-install.sh -h -r` command. Be sure to include the `-h -r` options when kicking off the build script for the build to work correctly.

```
dev@92fae72fafee:~/trs-workspace$ ./trs-install.sh -h -r
Using Yocto cache from host
Using reference from host
Downloading Repo source from https://gerrit.googlesource.com/git-repo
repo: Updating release signing keys to keyset ver 2.3
warning: gpg (GnuPG) is not available.
warning: Installing it is strongly encouraged.

repo has been initialized in /home/dev/trs-workspace
Fetching: 71% (10/14) Linaro/trusted-reference-stack/trs.git
...
```

Note: This build currently requires several hours to complete. There will be a number of warnings during the build, but this is OK. If completes successfully, then you'll see a message prior to returning to the prompt similar to the following:

```
Summary: There were 4 WARNING messages.
Build succeeded, see output in build/tmp_trs-qemuarm64/deploy directories.
dev@92fae72fafee:~/trs-workspace$
```

Once the build succeeds, the user can perform a final verification step, which is to execute the steps in the *Install QEMU* section of this document.

CHAPTER

FIVE

RUN ON BARE-METAL

This document describes how to run TRS for various supported targets.

The easiest way to get TRS up and running is

- Flash your device firmware to the correct medium
- Prepare a USB disk with the OS

5.1 Flashing the firmware

Firmware is device specific. As a result each of the supported boards has vendor specific requirements for writing the firmware.

You can find per device instructions in our *Installing firmware* section.

> **Warning:** If your firmware is going to be flashed on an SD card make sure the device in `/dev` is present before proceeding. If the `/dev/sdX` file is missing you will end up creating a static in `/dev` and write nothing on the SD card. The card will not be detected until you delete the file!

5.2 Prepare USB stick with TRS

To flash the rootfs image you built above, from your TRS build directory

```
$ sudo dd if=build/tmp_trs-qemuarm64/deploy/images/trs-qemuarm64/trs-image-trs-qemuarm64.
↪wic of=/dev/sdX bs=1M status=progress
$ sync
```

5.3 Boot TRS

Attach the USB stick on USB port and reset the device. If your USB stick is detected TRS will boot automatically.

> **Warning:** Always prefer USB 3.0+ ports. If you have problems booting TRS, interrupt U-Boot boot sequence and make sure your disk is detected.

```
=> usb start
=> usb storage
  Device 0: Vendor: SanDisk Rev: 1.00 Prod: Cruzer Blade
            Type: Removable Hard Disk
            Capacity: 29340.0 MB = 28.6 GB (60088320 x 512)
```

CHAPTER

SIX

INSTALL QEMU

This document describes how to run TRS for the QEMU target. It is assumed that you have completed the procedures outlined on the *Getting started* page and at least built the firmware for the `tsqemuarm64-secureboot` target and the `trs` image. If not, begin there before proceeding.

6.1 Run

After the build is complete, you will be able to run it on your host system using QEMU.

```
$ make run
```

U-Boot is already set to boot the current kernel, initramfs, and rootfs upon initial startup.

Note: To quit QEMU, press `Ctrl-A x` (alternatively kill the qemu-system-aarch64 process)

If everything goes as planned, you will be greeted with a login message and a login prompt. The login name is `ewaol` as depicted below.

```
ledge-secure-qemuarm64 login: ewaol
ewaol@ledge-secure-qemuarm64:~$
```

Alternatively if you want to launch QEMU manually follow the instructions *Run on QEMU arm64*

6.2 Test

Once the build has been completed, you can run automatic tests with QEMU. These boot QEMU using the compiled images and run test commands via SSH on the running system. While the QEMU image is running, SSH access to it works via localhost IP address `127.0.0.1` and TCP port `2222`. `TEST_SUITES` variable in `trs-image.bb` recipe define which tests are executed.

```
$ cd <workspace root>
$ make test
```

See Yocto runtime testing documentation for details about the test environment and instructions for writing new tests.

CHAPTER

SEVEN

TRS RECIPES

TRS levarage various layers and recipes to build firmware, root filesystem and various images. The best place to start looking for the recipes used, would be in the manifest files (`*.xml`) in trs-manifest.git.

CHAPTER

EIGHT

FAQ

8.1 My board only has an SD card

We boot the system using an SD card only. However, we need to merge firmware and root file system images into a single image and store it into the SD card. Luckily we provide a script for that:

```
$ gunzip <firmware image>.wic.gz
$ wget https://git.linaro.org/ci/job/configs.git/plain/ledge/ts/scripts/ts-merge-images.
↪sh
$ chmod +x ts-merge-images.sh
$ ./ts-merge-images.sh <firmware image>.wic trs-image-trs-qemuarm64.wic
```

Verify the images are programmed correctly. Note that only "ESP" and "Root Filesystem" will be identical on your board. The number and nature of the preceding are vendor specific.

Here is an example from a RockPI4b board:

```
$ fdisk -l ts-firmware-rockpi4b.wic
Disk ts-firmware-rockpi4b.wic: 2.28 GiB, 2443199488 bytes, 4771874 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: B9476BE0-8456-4A3B-98D4-75A91739819F

Device                      Start     End Sectors  Size Type
ts-firmware-rockpi4b.wic1      64    8063    8000  3.9M unknown
ts-firmware-rockpi4b.wic2    8064    8191     128   64K Microsoft basic data
ts-firmware-rockpi4b.wic3    8192   16383    8192    4M Microsoft basic data
ts-firmware-rockpi4b.wic4   16384   24575    8192    4M unknown
ts-firmware-rockpi4b.wic5   24576   32767    8192    4M Microsoft basic data
ts-firmware-rockpi4b.wic6   32768  557055  524288  256M EFI System         <------- ESP
ts-firmware-rockpi4b.wic7  557056 4751359 4194304    2G Linux filesystem   <-------␣
↪Root Filesystem
```

8.2 Q: How to increase OP-TEE core log level?

Add `CFG_TEE_CORE_LOG_LEVEL=3` to `EXTRA_OEMAKE` in `meta-ts/meta-arm/recipes-security/optee/optee-os.inc` and rebuild (kas build…)

8.3 Q: How to modify optee-os sources locally and rebuild?

1. Remove line `INHERIT += rm_work` in `ci/base.yml`
2. **Run `$ kas shell ci/rockpi4b.yml`**
    1. `bitbake -c cleansstate optee-os` # WARNING removes source in work directory
    2. `$ bitbake optee-os`
    3. Edit source files in build/tmp/work/rockpi4b-poky-linux/optee-os/<ver>/git `$ bitbake -c compile -f optee-os` # mandatory before kas build below it seems
3. Exit kas shell and run `$ kas build ci/rockpi4b.yml`

8.4 Q: Why is the internal eMMC not detected?

Try a different USB-C power supply. We use a Dell one. I have another no-name PS supposedly rated PD 100W which doesn't work reliably.

8.5 Q: How to skip initramfs and boot to rootfs directly?

```
$ efidebug boot add -b 1 TRS usb 0:1 Image -s 'panic=60 root=/dev/sda2 rootwait'; efidebug boot order 1; bootefi bootmgr
```

8.6 Q: On boot, the kernel logs warnings about GPT, how to fix them?

They are harmless, they are caused by the fact that the actual device (USB key) is larger than the image copied to it. The warnings can be removed by running `gparted /dev/sdaX` and accepting the prompt to fix the GPT info.

8.7 Q: On boot, the kernel logs "EXT4 … recovery complete", what's wrong?

Usually harmless. The board was not powered off or rebooted cleanly. Use `systemctl halt` or `systemctl reboot`.

8.8 Q: symbolize.py for TAs (on e.g., the fTPM TA) prints DWARF warnings and no source file/line info.

The default toolchains (`aarch64-linux-gnu-*`) is too old (7.2). Put a more recent one in your `PATH` before invoking `symbolize.py` (Note: some source/file line info are still missing, could be due to build flags)

8.9 Q: My board randomly hangs or crashes under system load.

Some boards are very picky about their PSU. Ensure you are using an official PSU. E.g for the RockPI4b https://shop.allnetchina.cn/products/power-supply-adapter-qc-3-0-for-rock-pi-4

Do not use a 5v only USB-C PSU (such as a USB port on your laptop), as you will hit random board stability issues.

CHAPTER

NINE

FIRMWARE

9.1 Trusted Substrate

Trusted Substrate is a meta-layer in OpenEmbedded aimed towards board makers who want to produce an Arm SystemReady (based on [EBBR]) compliant firmware and ensure a consistent behavior, tamper protection and common features across platforms. In a nutshell TrustedSubstrate is building firmware for devices which verifies the running software hasn't been tampered with. It does so by utilizing a well known set of standards.

- **UEFI secure boot enabled by default**

  UEFI Secure Boot is a verification mechanism for ensuring that code launched by a computer's UEFI firmware is trusted. It is designed to protect a system against malicious code being loaded and executed early in the boot process, before the operating system has been loaded.

- **Measured boot. With a discrete or firmware TPM**

  Measured Boot is a method where each of the software layers in the boot sequence of the device , measures the next layer in the execution order, and extends the value in a designated TPM PCR. Measured boot further validates the boot process beyond Secure Boot.

- **Dual banked firmware updates with rollback and bricking protection**

  Dual banked firmware updates provides protection to the firmware update mechanism and shield the device against bricking as well as rollback attacks.

9.2 Hardware and Software

9.2.1 Supported Platforms

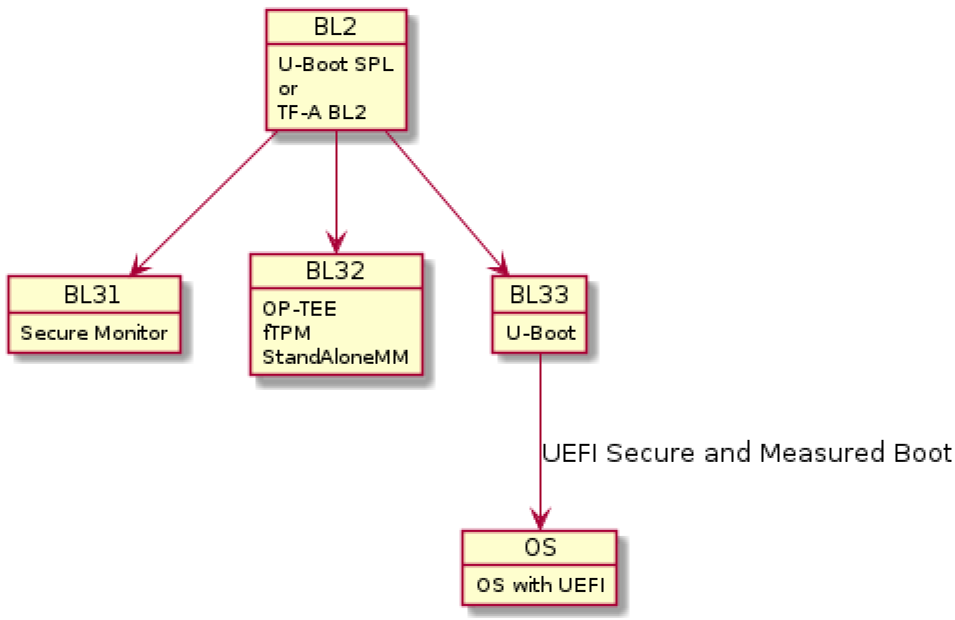
Trusted Substrate supports a variety of armv8 and armv7 boards. It's important to understand that the hardware characteristics dictate the supported features as well as the level of the device security

Software Components

Generally the following software components are used to boot up the boards and setup the chain of trust

- U-Boot
- OP-TEE
- TF-A
- firmware TPM
- StandAloneMM from EDK2
- SCP

A high level overview of the boot chain looks will look like this



Board Support

- QEMU (arm64)
- SynQuacer DeveloperBox
- stm32mp157c-dk2
- stm32mp157c-ev1
- Rockpi4
- Raspberry Pi4
- Xilinx kv260 starter kit
- Xilinx kv260 commercial

Supported platform features

Board	FSBL	Secure Boot	Measured Boot	Auth. Capsule Updates	A/B updates
QEMU	TF-A	Yes (Built-in vars)	Yes	No	No
DeveloperBox	SCP + TF-A	Yes (RPMB vars)	Yes [fTPM]	Yes	WIP
stm32mp157c-dk2	TF-A	Yes (Built-in vars)	No	No	WIP
stm32mp157c-ev1	TF-A	Yes (RPMB vars)	No	No	WIP
Rockpi4	U-Boot SPL	Yes (RPMB vars)	Yes [fTPM]	Yes	No
Raspberry Pi4	Proprietary	Yes (Built-in vars)	Yes (needs SPI TPM)	No	No
Xilinx kv260 starter kit	U-Boot SPL	Yes (Built-in vars)	Yes	Yes	WIP
Xilinx kv260 commercial	U-Boot SPL	Yes (Built-in vars)	Yes	Yes	WIP

9.3 Build and install

9.3.1 Getting the firmware

Building from source

Trusted Substrate depends on a couple of different packages being present in the host OS environment to be able to successfully build the firmware. The list of packages known to be needed can be found below.

Prerequisites for meta-ts

Python packages:

```
pip install kas
```

Debian based distro packages :

```
sudo apt install chrpath diffstat lz4
```

Building meta-ts from source

Compiling for different boards is straightforward.

> **Warning:** Since UEFI secure boot is enabled by default, boards that embed the UEFI keys in the firmware binary will use the predefined Linaro certificates. Those boards will only be allowed to boot images signed by the afforementioned Linaro certificates.
>
> *Building with your own certificates* if you want to generate your own
>
> *Hardware and UEFI variable limitations* for hardware limitations

```
git clone https://gitlab.com/linaro/trustedsubstrate/meta-ts.git
cd meta-ts
kas build ci/<board>.yml
```

replace <board> with

- qemuarm64-secureboot
- synquacer
- stm32mp157c-dk2
- stm32mp157c-ev1
- rockpi4b
- rpi4
- zynqmp-kria-starter

The build output is in `build/tmp/deploy/images/`

> **Hint:** The build directory contains a lot of artifacts. Look at *Installing firmware* for the per board files you need

Downloading board binaries

We do produce daily builds for all the support boards here

Building with your own certificates

> **Warning:** The default nightly builds we provide for devices that embed the keys are using a private key that is available at `meta-trustedsubstrate/uefi-certificates/`. Anyone could sign and boot an EFI binary! **This is a mandatory step for a production firmware!**

You need to generate the following keys:

- PK - Platform Key (Top-level key)
- KEK - Key Exchange Keys (Keys used to sign Signatures Database and Forbidden Signatures Database updates)
- db - Signature Database (Contains keys and/or hashes of allowed EFI binaries)

- dbx - Forbidden Signature Database (Contains keys and/or hashes of forbidden EFI binaries)

Refer to *Create certificates and keys* for generating certificates and create tar.gz archive with the .esl files

```
tar -czf uefi_certs.tgz db.esl dbx.esl KEK.esl PK.esl
```

Set up an environment variable `UEFI_CERT_FILE: "<path>/uefi_certs.tgz"` in your `local.conf` or in `ci/base.yml` and recompile your firmware.

Note: This is **only** needed if the variables are built-in into the firmware binary. You don't need this if your board has an RPMB and OP-TEE support.

9.3.2 Installing firmware

If your hardware can boot of an SD-card meta-ts will generate a WIC image which you can `dd` to your target. Otherwise the firmware must be flashed in a board specific way.

Since the firmware provides a [UEFI] interface you are free to choose the distro you prefer.

QEMU arm64

QEMU just needs the build file containing all the firmware binaries.

Note: Files needed from build directory **flash.bin**

SynQuacer

The SynQuacer can't boot from an SD card. You need to download and install the firmware via `xmodem`. You can find detailed instructions here

The short version is flip DSW2-7 to enable the serial flasher, open your minicom and use `xmodem` to send and update the files.

```
flash write cm3 ->  Control-A S   (send scp_romramfw_release.bin)
flash rawwrite 0x600000 0x200000  (Control-A S -> fip.bin)
```

After successful firmware update via serial flasher, power off the board, set DSW2-7 to OFF, DSW3-3 and DSW3-4 to ON to enable OP-TEE and TBB(Trusted Board Boot).

Note: Files needed from build directory **scp_romramfw_release.bin**, **fip.bin**

stm32mp157c dk2 or ev1

```
zcat ts-firmware-stm32mp157c-dk2.wic.gz > /dev/sdX
zcat ts-firmware-stm32mp157c-ev1.wic.gz > /dev/sdX
```

Note: Files needed from build directory **ts-firmware-stm32mp157c-dk2.wic.gz** or **ts-firmware-stm32mp157c-ev1.wic.gz**

rockpi4b

```
zcat ts-firmware-rockpi4b.rootfs.wic.gz > /dev/sdX
```

Note: Files needed from build directory **ts-firmware-rockpi4b.rootfs.wic.gz**

Raspberry Pi4

```
zcat ts-firmware-rpi4.wic.gz > /dev/sdX
```

Note: Files needed from build directory **ts-firmware-rpi4.wic.gz**

Xilinx KV260 AI Starter kit

This board uses an internal SPI flash. You need to reset the board while pressing `FWUEN` switch. This will launch an HTTP server at `192.168.0.111`

Connect to the web Interface and update ImageA and ImageB

Note: Files needed from build directory **ImageA.bin**, **ImageB.bin**

9.3.3 Updating the firmware

Generating capsules

Capsules will automatically be built along with the firmware files. You can find them in the boards build directory *build/tmp/deploy/images/<machine>/<machine>_fw.capsule*

Applying capsules from the command line

- Copy the capsules in the ESP in the `\EFI\UpdateCapsule` directory
- Since the `\EFI\UpdateCapsule` is only checked for capsules within the device that an active boot option is specified, make sure your `BootOrder` variables are correctly set. Alternatively tou can set `BootNext` variable with (assumin the capsule is on your mmc) `efidebug boot add -b 1001 cap mmc 1:1 EFI/UpdateCapsule && efidebug boot next 1001`
- In U-Boot console issue `setenv -e -nv -bs -rt -v OsIndications =0x0000000000000004`
- Reboot the board the capsules should be detected and applied. Alternatively you can manually apply the capsules with `efidebug capsule disk-update` using the U-Boot console.

If processing the capsule is sucessful you should see something like the following in the log.

```
Applying capsule <capsule file> succeeded
Reboot after firmware update
resetting ...
```

More information about capsules and uefi in U-Boot can be found U-Boot capsule update

Applying capsules from the OS

Capsule update-on-disk is supported via `fwupd`. When `fwupd` runs, it will copy the firmware files to `\EFI\UpdateCapsule` of the ESP. Once the board reboots capsule will be applied automatically. More information can be found here

TrustedSubstrate builds the required .cab files for all the platforms. You can find them in the build directory as `<machine name>_fw.cab`

```
sudo fwupdtool install /path/to/<machine name>_fw.cab
```

Note: The EFI Spec mandates: *The directory EFIUpdateCapsule is checked for capsules only within the EFI system partition on the device specified in the active boot option determined by reference to BootNext variable or BootOrder variable processing. The active Boot Variable is the variable with highest priority BootNext or within BootOrder that refers to a device found to be present. Boot variables in BootOrder but referring to devices not present are ignored when determining active boot variable.*

Since SetVariable at runtime is not yet supported, the only available option is place the EFIUpdateCapsule within the ESP partition indicated by the current BootOrder.

9.4 Configuration and OS booting

9.4.1 Configuring UEFI variables

Boards that embed the UEFI keys in the U-Boot binary *Hardware and UEFI variable limitations* won't allow you to change the EFI security related variables (PK, KEK, db and dbx).

That category of boards comes with a predefined set of keys. For more details look at *Building with your own certificates*

Enabling Secure Boot

Secure Boot is enabled and disabled automatically based on the existence of a Platform Key (PK). Enrolling one will enable UEFI Secure Boot and all the EFI binaries must to be signed.

For more details look at [UEFI] (§ 32.3.1 Enrolling The Platform Key)

Create certificates and keys

Copy and run the script below. The .auth files you need can be found in `efi_keys/` directory and the private certificates on `priv_keys`.

Note: This script is provided as sample. Always backup your SSL certificates directory!

```
#!/bin/bash
# sudo apt install efitools openssl uuid-runtime
set -e
CN='mytestCA'
OUT_DIR=priv_keys/
OUT_EFI_DIR=efi_keys/

mkdir $OUT_DIR -p
mkdir $OUT_EFI_DIR -p
if [ ! -e "$OUT_DIR/GUID.txt" ]; then
    GUID=$(uuidgen)
    echo $GUID > $OUT_DIR/GUID.txt
else
    echo "Please remove '"$OUT_DIR"GUID.txt' to regenerate certs"
    echo "This will overwrite your private keys!"
    exit 1
fi

for cert in PK KEK db dbx; do
    # SSL certs
    openssl req -new -x509 -newkey rsa:2048 -subj "/CN=$CN $cert/" -keyout \
        $OUT_DIR/$cert.key -out $OUT_DIR/$cert.crt -days 3650 -nodes -sha256

    # EFI signature list certs
    # .esl certs can be concatenated if we want to support multiple signers
    cert-to-efi-sig-list -g $GUID $OUT_DIR/$cert.crt $OUT_EFI_DIR/$cert.esl
done
# Empty PK to reset secure boot
rm -f $OUT_EFI_DIR/noPK.esl
touch $OUT_EFI_DIR/noPK.esl

sign-efi-sig-list -c $OUT_DIR/PK.crt -k $OUT_DIR/PK.key PK $OUT_EFI_DIR/noPK.esl $OUT_
↪EFI_DIR/noPK.auth
sign-efi-sig-list -c $OUT_DIR/PK.crt -k $OUT_DIR/PK.key PK $OUT_EFI_DIR/PK.esl $OUT_EFI_
↪DIR/PK.auth
sign-efi-sig-list -c $OUT_DIR/PK.crt -k $OUT_DIR/PK.key KEK $OUT_EFI_DIR/KEK.esl $OUT_
↪EFI_DIR/KEK.auth
```

(continues on next page)

(continued from previous page)

```
sign-efi-sig-list -c $OUT_DIR/KEK.crt -k $OUT_DIR/KEK.key db $OUT_EFI_DIR/db.esl $OUT_
↪EFI_DIR/db.auth
sign-efi-sig-list -c $OUT_DIR/KEK.crt -k $OUT_DIR/KEK.key dbx $OUT_EFI_DIR/dbx.esl $OUT_
↪EFI_DIR/dbx.auth
chmod 0600 $OUT_DIR/*.key
```

Enable Secure Boot

The commands below assume the keys are stored in the first partition of a usb stick.

```
load usb 0:1 90000000 PK.auth && setenv -e -nv -bs -rt -at -i 90000000:$filesize PK
load usb 0:1 90000000 KEK.auth && setenv -e -nv -bs -rt -at -i 90000000:$filesize KEK
load usb 0:1 90000000 db.auth && setenv -e -nv -bs -rt -at -i 90000000:$filesize db
load usb 0:1 90000000 dbx.auth && setenv -e -nv -bs -rt -at -i 90000000:$filesize dbx
```

Disable Secure Boot

The commands below assume the keys are stored in the first partition of a usb stick.

```
load usb 0:1 90000000 noPK.auth && setenv -e -nv -bs -rt -at -i 90000000:$filesize PK
```

9.4.2 Running a distro

Since the firmware provides a [UEFI] interface you are free to choose the distro you prefer. However boards that embed the UEFI keys in the U-Boot binary *Hardware and UEFI variable limitations* will only be able to boot signed binaries. Look at *Building with your own certificates* if you want to build and your own vertical distro and sign your binaries. If you use the pre-compiled firmware binaries you can test that with our own TRS distro.

Download TRS

Download a .wic.gz image from here

Running TRS

Throughout the examples we will be using a USB disk. If you prefer a different installation medium you need to adjust the commands accordingly.

You can prepare one with

```
zcat trs-image-trs-qemuarm64.rootfs.wic.gz > /dev/sdX
```

TRS comes with GRUB installed. As a result there is nothing else you have to do to boot your board. Just insert your USB disk and your device will automatically boot.

Note: TRS, on the first boot, will automatically encrypt your root filesystem if measured boot is enabled on your firmware.

Running TRS without GRUB

If you want to skip GRUB you need to configure the EFI boot manager properly.

Run on QEMU arm64

QEMU can provide a TPM implementation via Software TPM

[SWTPM] provides a memory mapped device which adheres to the TCG TPM Interface Specification

```
sudo apt install swtpm swtpm-tools

mkdir /tmp/mytpm1 -p

swtpm_setup --tpmstate /tmp/mytpm1 --tpm2 --pcr-banks sha256
swtpm socket --tpmstate dir=/tmp/mytpm1 \
    --ctrl type=unixio,path=/tmp/mytpm1/swtpm-sock \
    --log level=0 --tpm2 -t -d
```

```
gunzip trs-image-trs-qemuarm64.rootfs.wic.gz
qemu-system-aarch64 -m 2048 -smp 2 -nographic -cpu cortex-a57 \
    -bios flash.bin -machine virt,secure=on \
    -drive id=os,if=none,file=trs-image-trs-qemuarm64.rootfs.wic \
    -device virtio-blk-device,drive=os \
    -chardev socket,id=chrtpm,path=/tmp/mytpm1/swtpm-sock \
    -tpmdev emulator,id=tpm0,chardev=chrtpm \
    -device tpm-tis-device,tpmdev=tpm0
```

```
=> efidebug boot add -b 1 TRS virtio 0:1 Image -i virtio 0:1 ledge-initramfs.rootfs.cpio.
↪gz -s 'root=UUID=6091b3a4-ce08-3020-93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

Run on SynQuacer

```
=> efidebug boot add -b 1 TRS usb 0:1 Image -i usb 0:1 ledge-initramfs.rootfs.cpio.gz -s
↪'root=UUID=6091b3a4-ce08-3020-93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

Run on stm32mp157c dk2 or ev1

TRS does not yet provice Armv7 builds. Command for reference

```
=> efidebug boot add -b 1 TRS usb 0:1 Image -i usb 0:1 ledge-initramfs.rootfs.cpio.gz -s
↪'root=UUID=6091b3a4-ce08-3020-93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

run on rockpi4b

```
=> efidebug boot add -b 1 TRS usb 0:1 Image -i usb 0:1 ledge-initramfs.rootfs.cpio.gz -s
↪'root=UUID=6091b3a4-ce08-3020-93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

Run on Raspberry Pi4

```
=> efidebug boot add -b 1 TRS usb 0:1 Image -i usb 0:1 ledge-initramfs.rootfs.cpio.gz -s
↪'root=UUID=6091b3a4-ce08-3020-93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

Run on Xilinx KV260 AI Starter and Commercial kit

USB is not yet supported in the kernel. Use the mmc interface instead

```
=> efidebug boot add -b 1 TRS mmc 0:1 Image -i mmc 0:1 ledge-initramfs.rootfs.cpio.gz -s
↪'root=UUID=6091b3a4-ce08-3020-93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

9.5 References

9.6 Terms and abbreviations

This document uses the following terms and abbreviations.

UEFI Unified Extensible Firmware Interface.

EBBR Embedded Base Boot Requirements

FSBL First stage boot loader

TPM Trusted Platform Module

PK Platform Key

KEK Key Exhange Key

db Signature Database

dbx Forbidden Signature Database

ESP EFI System Partition

RPMB Replay Protected Memory Block

TCG Trusted Computing Group

CHAPTER

TEN

FEATURES

10.1 Secure Boot

The firmware component of TRS unconditionally enables UEFI secure boot for all supported platforms. There are some hardware requirements that will dictate how Secure Boot is configured and enabled on your hardware.

[UEFI] (§ 32.3.6 Platform Firmware Key Storage Requirements) defines that the Platform and Key exchange keys must be stored in a non-volatile storage which is tamper protected.

On Arm servers this is usually tackled by having a dedicated flash which is only accessible by the secure world.

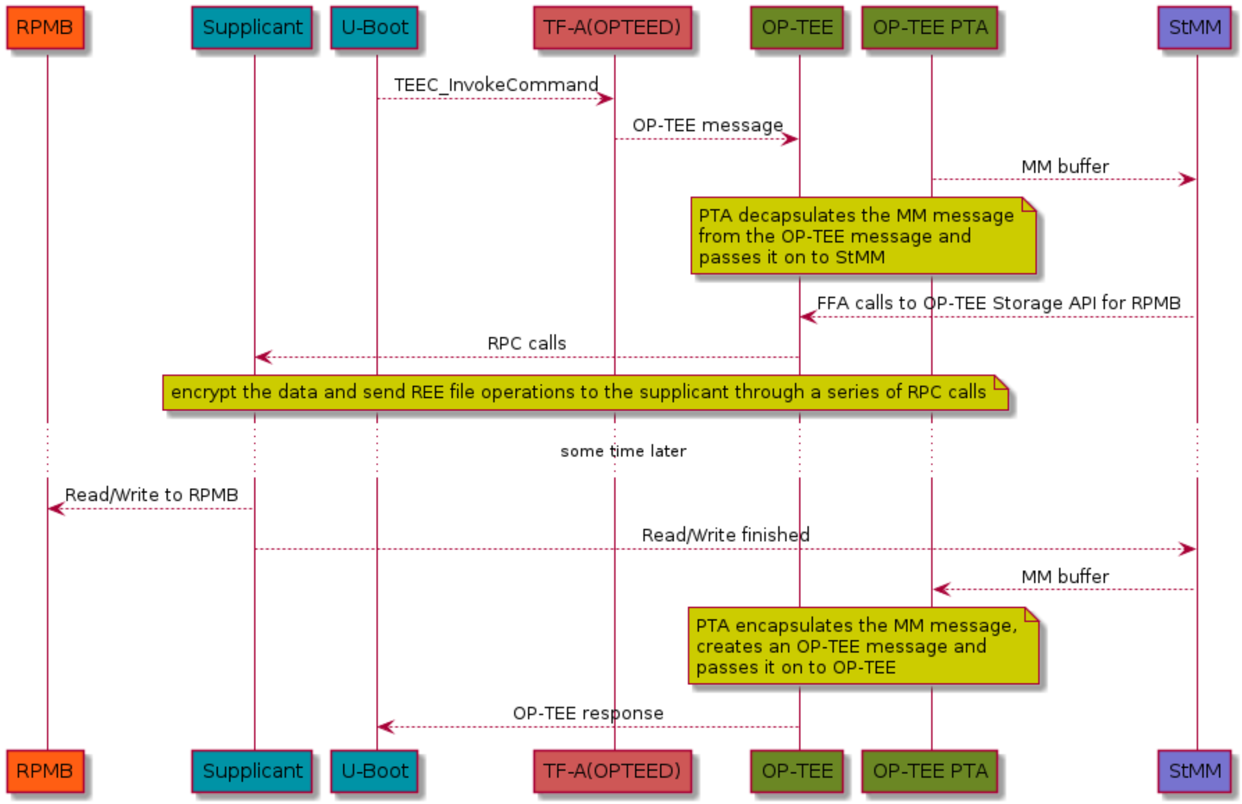
Hardware which was designed with security in mind has the following options.

Hardware	UEFI Secure Boot	Measured Boot
RPMB[1]	x	x
Discrete TPM		x
Flash in secure world	x	

The reality on embedded boards is different though. In the embedded case, we don't have a dedicated flash. What's becoming more common though is eMMC devices with an RPMB partition.

If the board has a RPMB and OP-TEE support, Trusted Substrate will use that device to store all the EFI variables.

[1] Requires OP-TEE support and a way to program the RPMB with a unique per hardware key (e.g a fuse accessible only from the secure world). Setting EFI variables at runtime (from the OS) not supported

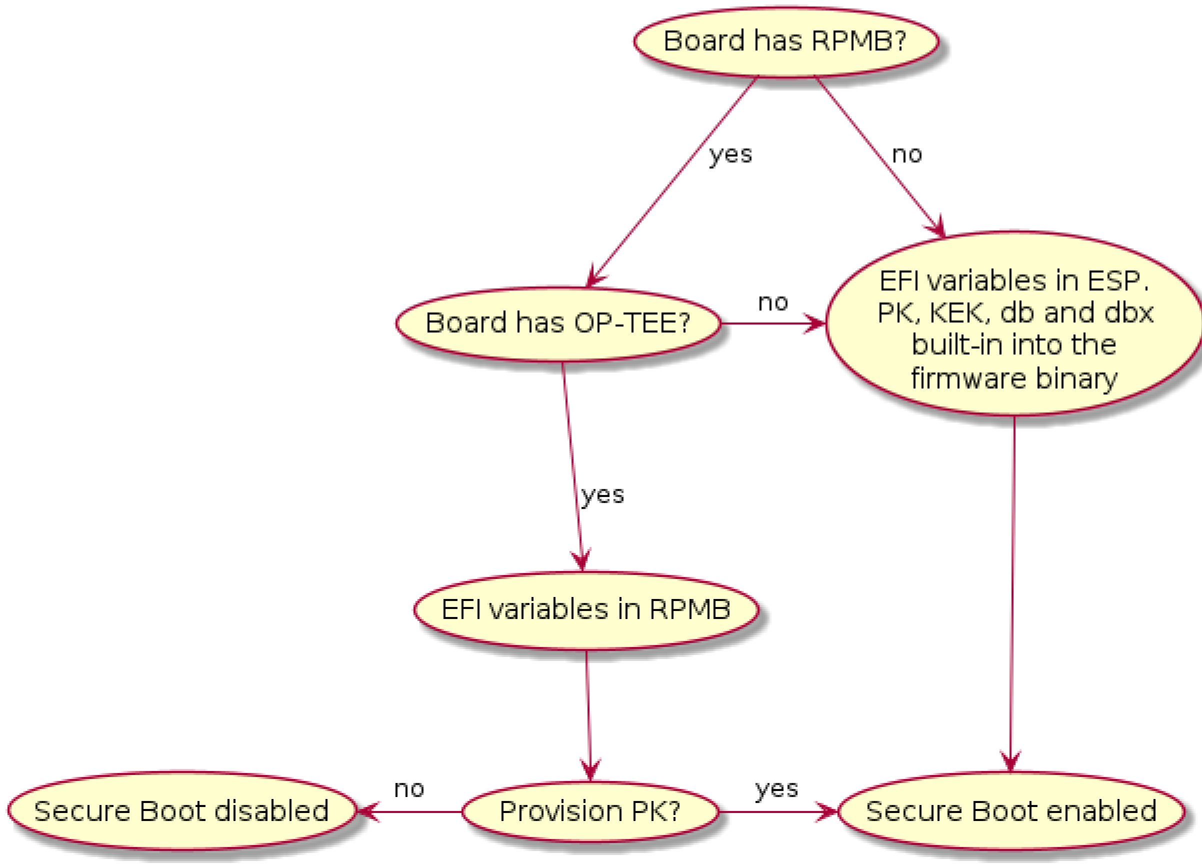


However for boards that don't have an RPMB the UEFI public keys (PK, KEK, DB etc) are built-in into the firmware binary. Bundling those keys comes with it's own set of limitations. The most notable ones being that in order to update any security related EFI variable, you need to update the bootloader and you can only boot signed binaries by default. Other, non security critical, EFI variables are stored in a file located in the ESP.

10.1.1 Hardware and UEFI variable limitations

The firmware automatically enables and disables UEFI Secure Boot based on the existence of the Platform Key (PK). As a consequence boards that embed the keys in the firmware binary will only be allowed to boot signed binaries and you won't be able to change the UEFI keys. See *Building with your own certificates*

On the other hand boards that store the variables in the RPMB come with an empty PK and the user must provision one during the setup process in order to enable Secure Boot.



10.2 Measured Boot

TRS is designed to take advantage of Trusted Platform modules. The firmware part of TRS supports the EFI TCG Protocol as well as TCG PC Client Specific Platform Firmware Profile Specification and provides the building blocks the OS needs for Measured Boot.

In TRS the software components that extend measurements are

- TF-A (QEMU only), creates an EventLog, which U-Boot will later replay on the TPM.
- U-Boot will measure all components described by the afforementioned TCG specs.
- The Linux kernel EFI-stub will measure the loaded initramfs and the EFI LoadOptions.

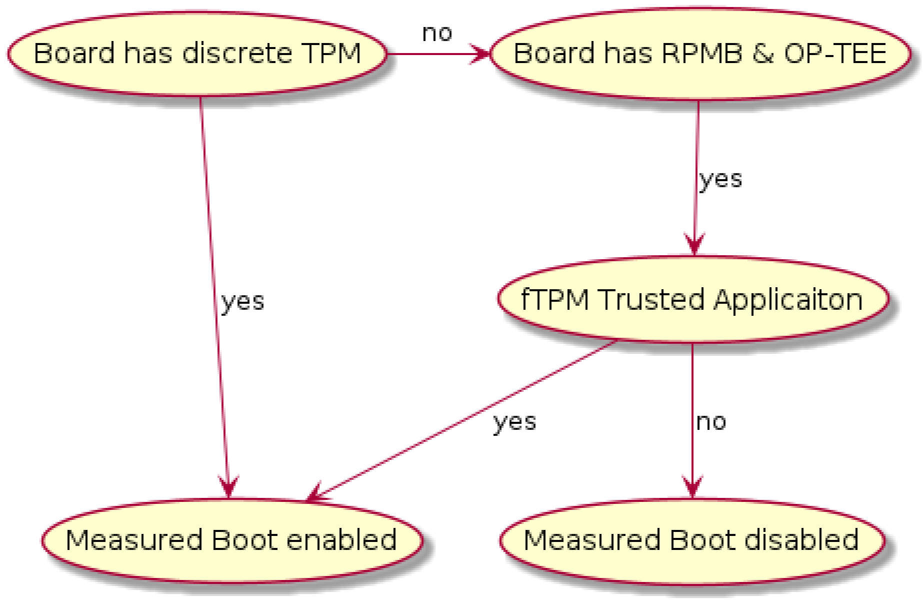
Note: PCR7 contains the UEFI Secure Boot keys and state. PCR9 will differ depending on your kernel version. Prior to 5.18 PCR9 will be empty. Past 5.18 and prior to 6.1 PCR9 will contains the initrd measurement. Post 6.1 it will contain the initrd and EFI LoadOptions measurements.

10.2.1 Trusted Platform module

TPMs are microcontrollers designed for cryptographic tasks. They contain a set of Platform Configuration Registers (PCRs) which are used to measure the system configuration and software.

PCRs start zeroed out and can only reset with a system reboot. Those can be extended by writing a SHA hash (typically SHA-1/256/384/512 for TPMv2) into the PCR. To store a new value in a PCR, the existing value is extended with a new value as follows: PCR[N] = HASHalg(PCR[N] || ArgumentOfExtend)

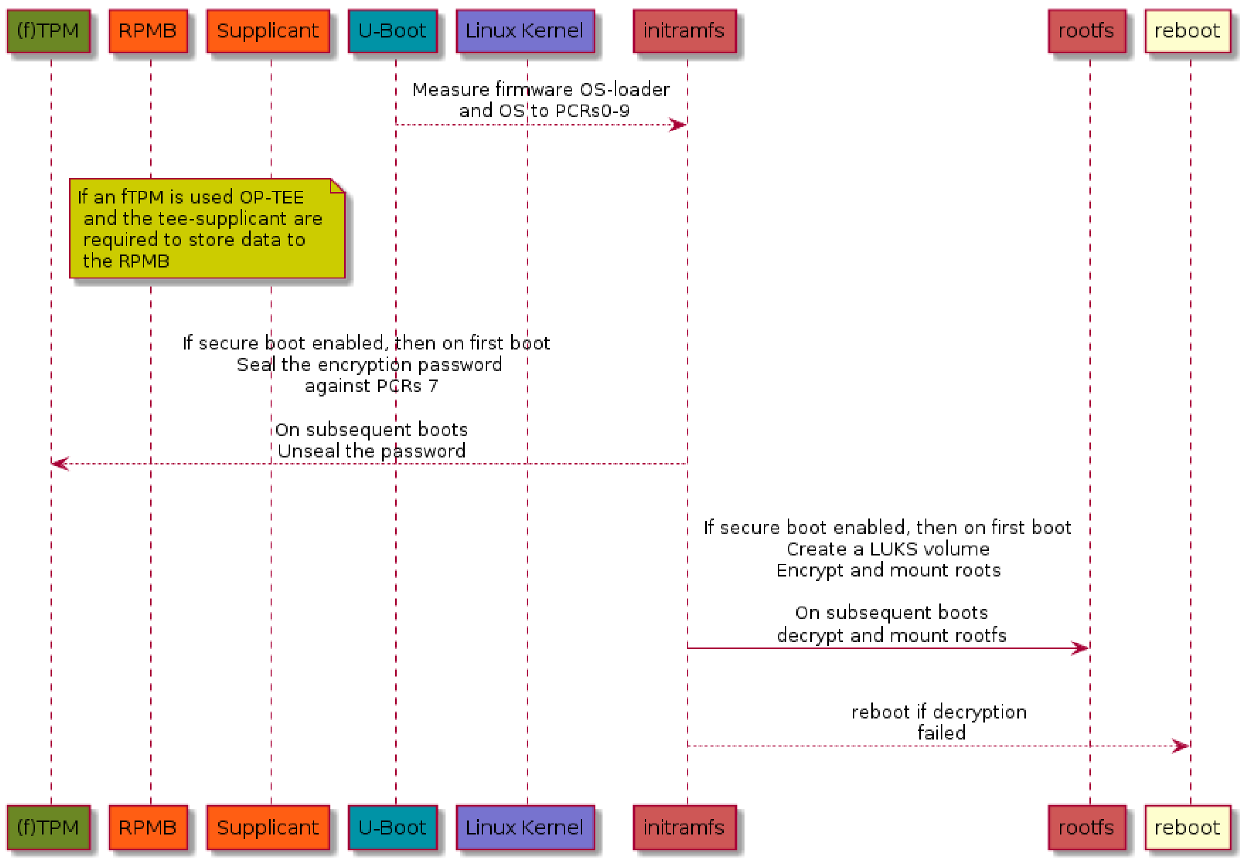
Trusted Substrate is designed to work with either discrete TPMs or provide an [fTPM] running in OP-TEE.



10.3 LUKS2 disk encryption

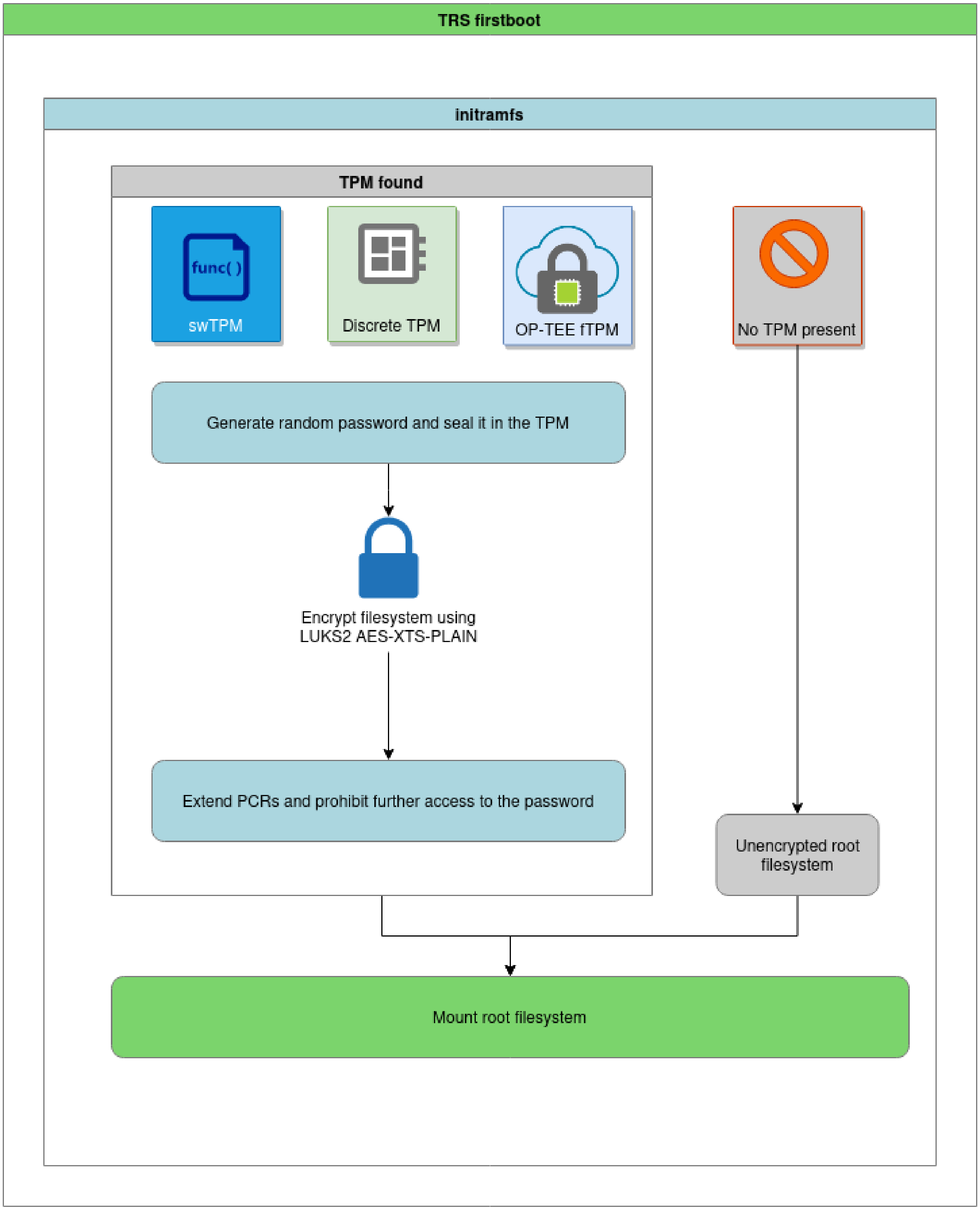
If a TPM is present on the device TRS will automatically detect it. If secure boot is enabled, then TRS will generate a random password on first boot, seal it against PCRs 7 and encrypt the root filesystem using aes-xts-plain.

TRS is designed to work regardless of the TPM implementation. We support devices with a discrete TPM, an [fTPM] or for QEMU a [SWTPM]

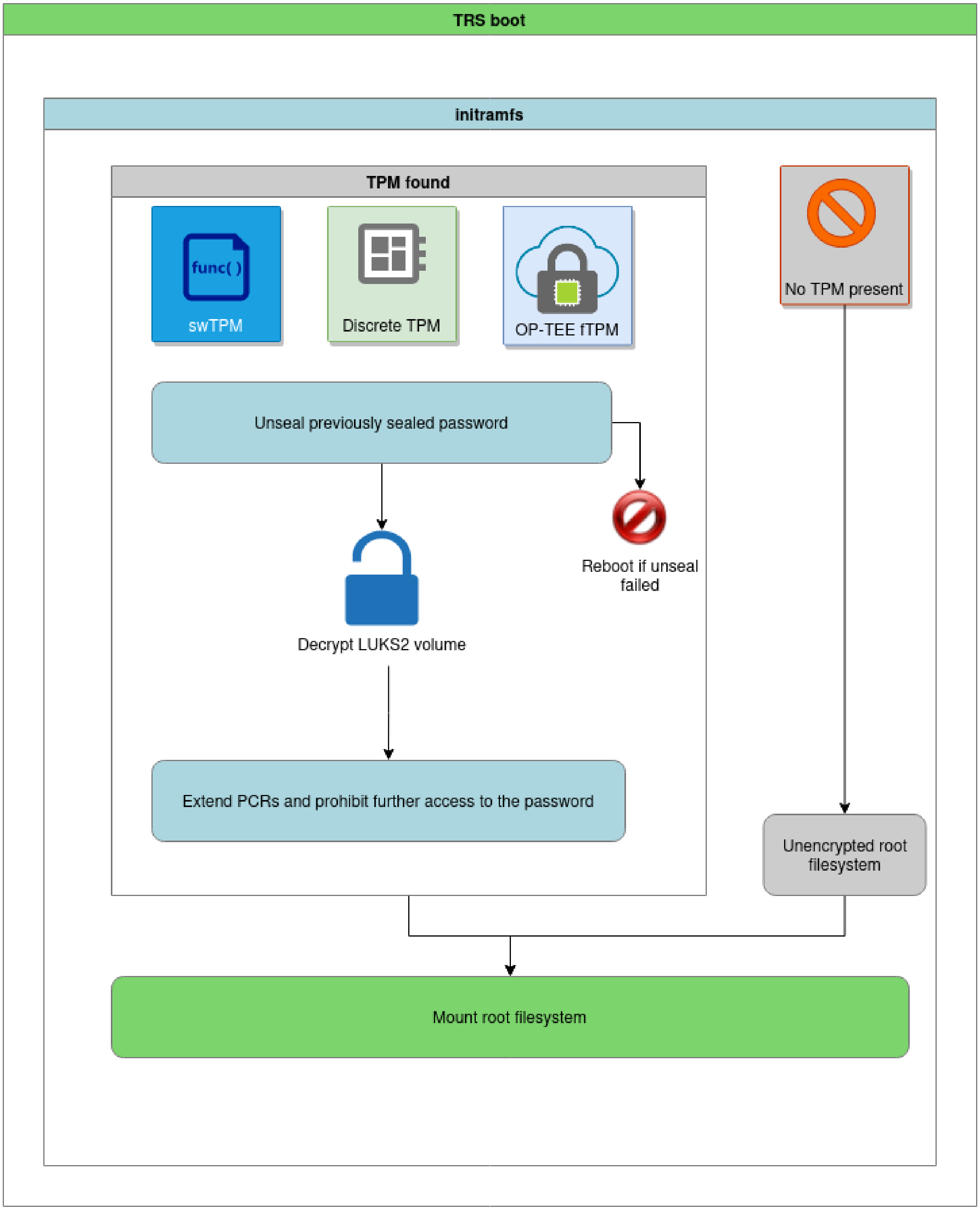


Note: You can find a full list of the components and recipes needed by running `make find name=ledge-initramfs`

10.3.1 LUKS2 Encryption



10.3.2 LUKS2 Decryption

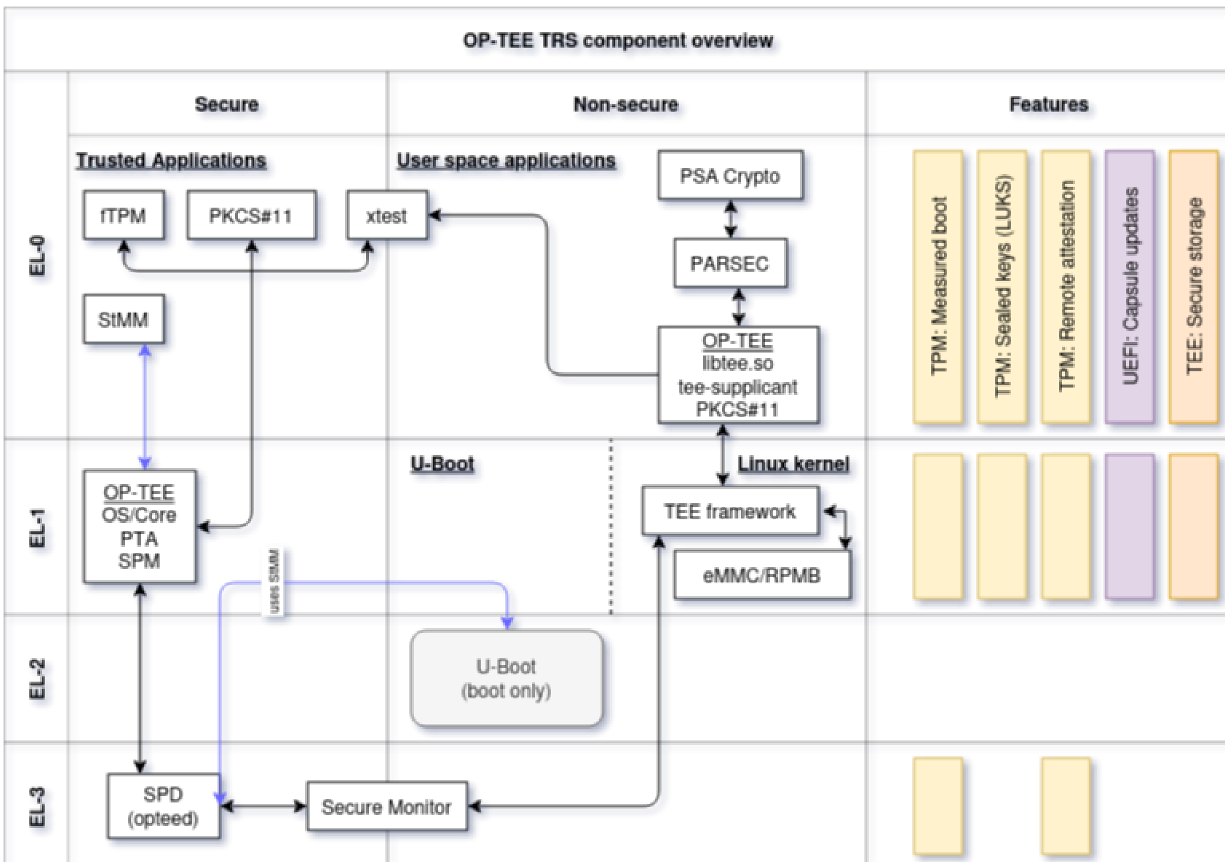


10.4 OP-TEE OS

OP-TEE is our Secure World OS of choice in TRS. We use it for a number of reasons with the most notable ones being

- Run [fTPM] is the hardware doesn't have a discrete TPM.
- Store EFI variables on boards that have an RPMB.
- Provide a DRBG if the hardware doesn't provide a TRNG.
- Provide a PKCS#11 provider to PARSEC.

Conceptually the components interacting with OP-TEE in the TRS build can be seen in the image below. The `Features` lane there indicates which exceptions levels are involved in a certain use case. For example, "TEE: Secure Storage" is all kept in (S)EL-0 and (S)EL-1.



Note that this image is rather generic as depicted here. We have other areas that could (and should) be added as well, for example `SCMI`, `Xen`, `FF-A`, `SwTPM` to name a few. But perhaps it's better to add them as separate diagrams to avoid making the images too complex.

10.5 Xen

When Xen is enabled, GRUB menu provides an entry *TRS Xen (if supported)* for booting Xen hypervisor.

```
                          GNU GRUB  version 2.11

/----------------------------------------------------------------------------\
| TRS                                                                        |
|*TRS Xen (if supported)                                                     |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
|                                                                            |
\----------------------------------------------------------------------------/

     Use the ^ and v keys to select which entry is highlighted.
     Press enter to boot the selected OS, `e' to edit the commands
     before booting or `c' for a command-line. ESC to return previous
     menu.
```

Xen hypvervisor's EFI program and configuration file (xen.cfg) both are placed in the root folder of boot parition. The configuration file contains the info for Xen's log debugging level, Linux kernel image path and Linux kernel command line, etc; Xen hypervisor parses the configuration file and boot Linux kernel image.

Note, Xen hypervisor doesn't load initial ramdisk, this is different from the booting flow in bare metal mode which loads both initial ramdisk and Linux kernel image.

```
# SPDX-License-Identifier: MIT

[global]
default=xen

[xen]
options=noreboot dom0_mem=4096M bootscrub=0 iommu=on loglvl=error guest_loglvl=error
kernel=Image console=hvc0 earlycon=xenboot rootwait root=PARTUUID=f3374295-b635-44af-
↪90b6-3f65ded2e2e4
```

After the system booting up, we can use the command *xl list* to list Xen domains, the Xen Dom0 with naming *Domain-0* is created by default.

```
root@trs-qemuarm64:~# xl list
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0  4096    32     r-----      63.2
```

At this time the goal is to use the same rootfs when booting Dom0 and DomU. The root file system in Xen Dom0 doesn't contain anything for Xen DomU, otherwise, we will run into the nested issue for building TRS image. For this reason, we need to take several steps to deploy virtual machine with Xen DomU, below gives instructions for how to do it.

Firstly, you need to create a virtual machine configuration file *ewaol-guest-vm1.cfg*:

```
# Copyright (c) 2022, Arm Limited.
#
# SPDX-License-Identifier: MIT

name = "ewaol-guest-vm1"
memory = 6144
vcpus = 4
extra = " earlyprintk=xenboot console=hvc0 rw"
root = "/dev/xvda2"
kernel = "/boot/Image"
disk = ['format=qcow2, vdev=xvda, access=rw, backendtype=qdisk, target=/usr/share/guest-
↪vms1/trs-vm-image.rootfs.wic.qcow2']
vif = ['script=vif-bridge,bridge=xenbr0']
```

The configuration file *ewaol-guest-vm1.cfg* can be saved into the folder */etc/xen/auto/* so the virtual machine can be automaticially launched in later's booting.

Secondly, we need to copy TRS root file system image to target. In below example, we firstly create a folder */usr/share/guest-vms1/* on the target:

```
root@trs-qemuarm64:~# mkdir -p /usr/share/guest-vms1/
```

Then we copy TRS's qcow2 image from the host to the target, please replace <IP_ADDRESS> with your target's IP address.

```
$ cd trs-workspace/build/tmp_trs-qemuarm64/deploy/images/trs-qemuarm64
$ scp trs-image-trs-qemuarm64.wic.qcow2 root@<IP_ADDRESS>:/usr/share/guest-vms1/trs-vm-
↪image.rootfs.wic.qcow2
```

No need to copy kernel image, the virtual machine can reuse the same kernel image with the Xen Dom0 which has been already placed in */boot/Image*.

With above preparations, it's ready for luanching the virtual machine in Xen domU. We can create a virtual machine with command:

```
root@trs-qemuarm64:~# xl create /etc/xen/auto/ewaol-guest-vm1.cfg
```

After created the virtual machine, we can list all Xen domains:

```
root@trs-qemuarm64:~# xl list
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0  4096    32     r-----      63.2
ewaol-guest-vm1                              1  6143     4     r-----       4.5
```

We can see a new domain *ewaol-guest-vm1* running in Xen DomU (ID is 1 with 4 virtual CPUs).

For accessing a Xen DomU's console, you could use the command *xl console* followed by a domain name, below is an example:

```
root@trs-qemuarm64:~# xl console ewaol-guest-vm1
```

Afterwards, you could input *ctrl-[* to exit from Xen DomU's console and return back to Xen Dom0.

Known issue1: Currently Xen hypervisor is only supported for ADLink AVA platform.

Known issue2: Xen hypvervisor loads kernel image but it doesn't load initial ramdisk.

Known issue3: TPM is not supported by Xen Dom0. If the system runs into the normal booting flow with GRUB menu entry *TRS*, the root file system image will be encrypted with TPM; afterwards when we switch back to Xen, it cannot reuse the root file system image due to Xen not supporting TPM at the current stage.

CHAPTER

ELEVEN

THREAT MODELS

We're leveraging the MITRE D3FEND threat model matrix as a basis for the threat modeling work in the TRS. Although MITRE D3FEND is more aimed at regular PC use, we believe it is a good and comprehensive summary of potential attacks to a lot of use cases in TRS. MITRE D3FEND covers the generic type of threats. In addition to that we will also identify the specific threats based on the assets that we're trying to protect. Re-use is key here, the first use-cases that we implement will cover quite a bit of mitigation techniques. For new use cases we anticipate that these should be able to leverage mitigations already implemented for other use cases.

11.1 Use cases

11.1.1 1. Attested containers

Assets

Table 1: Assets in attested containers

Asset	Description
Private key(s) used to sign the container images.	Private keys will be used to sign the container images.
Public key(s) used to verify signature.	Although not secret, they must be immutable in the system.
PCR registers in the TPM	They tell the true and expected state of a system.
Audit log files	Files under `/var/...` tracking events in the form of audit logs.
Authentication Tokens	When leveraging backends, it's common to get an authorization token from the backend provider.
Environment variables	Tokens and passwords sometimes needs to be stored in environment variables.
Kernel command line	Information provided via Linux kernel commandline could be vital (for the security of the system).
U-Boot commandline	Should be locked down on a production system to avoid system modification.

Hardening

Table 2: Threat model attested containers

Threat	Description	Mitigation
Insecure configuration (D3-ACH)	Software sometimes comes with default configurations that aren't secure.	• Follow the *TF-A*, *OP-TEE*, *TPM (fTPM)* recommended configurations for building a secure product. • Follow recommendations telling how to configure OCI-based containers for security oriented end products.
Physical access to configuration	The device can be deployed in a location where people have physical access to the device, which also means that they might try to change configurations.	• Boot time integrity checking of configurations. • Run-time integrity checking of configurations using for example IMA (D3-FH).
Bootloader Authentication	A legitimate user could try to replace or modify the firmware binaries.	• Signature verification using RSA or ECDSA. (D3-BA, D3-FV) • Measured boot (D3-TBI)
Corrupting memory	An attacker can try to modify memory to gain control of the execution (ROP, JOP attacks etc).	• Pointer Authentication (PAC) - requires Arm v8.3A. (D3-PAN) • Branch Target Identification (BTI) - requires Arm v8.5A. • Memory Tagging Extension (MTE) - requires Arm v8.5A. • Stack Frame Canary Validation (D3-SFCV) using for example GCC and `-fstack-protector`. • ASLR (D3-SAOR) to randomize base addresses.
Disk modification	An attacker physically move a disk or boot the machine in another OS and then try to alter the content on the disk.	• Disk Encryption (D3-DENCR).
Containers accessing host resources	Containers can run with elevated privleges, which can affect the security of the system.	• Avoid using `--privileged`, but at least document when using it and state why it is needed and what potential risks are. • Enable Mandatory Access Control (MAC) in form of Seccomp, SELinux etc. • Leverage `cgroups` to limit the access to system resources.
Container modification	An attack can try to replace or modify the container.	• Sign and verify containers (also see `podman image trust`, `podman image sign`)

11.2 Other projects threat models

11.2.1 TF-A

TrustedFirmware-A (TF-A) gives its analysis for threat model (ARM-TFA-THREAT-MODEL) and provides insecure configurations to mitigate potential threat. TRS suggests to enable these insecure configurations for a production ready build, the relevant flags are listed as below.

Table 3: TrustedFirmware-A (TF-A) insecure configurations

Inscure configurations	Description
ENABLE_STACK_PROTECTOR=strong	Enable the stack protection checks in GCC, the stack protection level "strong" is suggested.
BRANCH_PROTECTION=1	Enable the branch protection feature, setting to 1 means "Enables all types of branch protection features", it requires ARMv8.3 Pointer Authentication and ARMv8.5 Branch Target Identification are supported. Otherwise, if the CPUs on your platform cannot support one or both of these two CPU features, you need to select other values or event disable branch protection with setting value to 0. The detailed information can be found in the document ARM-TFA-BUILD-OPTIONS. To be able to leverage and build this feature, two additional flags needs to be enabled: CTX_INCLUDE_PAUTH_REGS=1 and ARM_ARCH_MINOR, we must pick the value for ARM_ARCH_MINOR based on the CPU architecture version, e.g. when validate on QEMU aarch64 with support Armv8.5 architecture, we set ARM_ARCH_MINOR=5 for this case.
DECRYPTION_SUPPORT=aes_gcm	Select the authenticated decryption algorithm for firmware.
ENCRYPT_BL31=1	Enable encryption for BL31 firmware.
ENCRYPT_BL32=1	Enable encryption for BL32 firmware.
KEY_ALG=rsa / KEY_SIZE=4096	Select the RSA algorithm for the PKCS keys and signing keys and the key size is 4096. When the large key size (4096) is used instead of the default key size of 2048, the product is better protected.
MEASURED_BOOT=1 / EVENT_LOG_LEVEL=10 / TPM_HASH_ALG	Enables measured boot option MEASURED_BOOT=1 when a platform supports TPM, we can emulate TPM with the tool *swtpm* on QEMU platform, the details for enabling TPM on QEMU can be found in the document QEMU-TPM. Setting EVENT_LOG_LEVEL=10 for only printing out TPM error log. TPM are used not only by TF-A but also by bootloaders and operating systems, usually the TPM PCR bank algorithm is chosen by later bootloader, this is reason why TF-A needs to explicitly specify TPM hash algorithm (e.g. set TPM_HASH_ALG=sha256) which is chosed by later bootloader and avoid incompatible issue between them.
DRTM_SUPPORT=1	Enable Dynamic Root of Trust for Measurement (DRTM).

As a reference, the TF-A recipe (QEMU-AARCH64-RECIPE) will enable above insecure configurations for building booting images for QEMU aarch64.

11.2.2 OP-TEE

Invoking the TEE from a container

Containers can access the services provided by OP-TEE as long as:

- The OP-TEE client libraries (`` `optee-client` `` package) are installed in the container
- The `/dev/tee0` device is exposed to the container. With Docker, this is achieved via `--device /dev/tee0`. For example:

```
$ docker run -it --device /dev/tee0 <docker-image>
```

With such a configuration, only the client side is deployed in the container; all the other components of the TEE are on the host. This includes:

- The OP-TEE kernel driver
- The MMC RPMB kernel driver (when OP-TEE's `` `CFG_RPMB_FS` `` is enabled)
- The `tee-supplicant` process
- The files created in the host's root filesystem by `tee-supplicant` to provide storage for TEE persistent objects (when OP-TEE's `CFG_REE_FS` is enabled)
- The OP-TEE OS
- The Trusted Applications binaries (`` `*.ta` `` files)

More complex configurations are possible, for example:

- Running `tee-supplicant` in a container. For this `dev/teepriv0` has to be shared with the container via `--device /dev/teepriv0`. Only one instance of the supplicant process may be running at any given time, so the host instance has to be stopped before the container is started.
- Loading Trusted Application from a container or moving secure storage into a container. `tee-supplicant` loads TAs from `/lib/optee_armtz` and manages data files for secure storage in `/data/tee` by default. Therefore, Docker bind mounts as well as host overlay mounts may be used to compose things in a creative way.

11.2.3 U-Boot

Unlike TF-A, U-Boot doesn't give any offical documentation for handling potential threats. Below lists insecure configurations which are suggested by TRS for a production ready build.

Table 4: U-Boot insecure configurations

Insecure configurations	Description
CONFIG_TPM / CONFIG_EFI_TCG2_PROTOCOL / CONFIG_EFI_TCG2_PROTOCOL_EVENTLOG_SIZE	Support TPM device on the platform, and enabling EFI_TCG2 configurations to produce EventLog with the TPM.
CONFIG_TEE / CONFIG_RNG_OPTEE	Enable driver for OP-TEE and create connection with secure world's OP-TEE firmware. Enable the OP-TEE based Random Number Generator.
CONFIG_EFI_RUNTIME_UPDATE_CAPSULE / CONFIG_EFI_CAPSULE_FIRMWARE / CONFIG_EFI_CAPSULE_FIRMWARE_RAW / CONFIG_EFI_CAPSULE_FIRMWARE_FIT	With these configurations, we can update the U-Boot image using the UEFI firmware management protocol (fmp). Enable CONFIG_EFI_CAPSULE_FIRMWARE_FIT to support FIP image with using the same protocol.
CONFIG_CMD_EFICONFIG / CONFIG_CMD_BOOTMENU / CONFIG_AUTOBOOT_MENU_SHOW / CONFIG_BOOTMENU_DISABLE_UBOOT_CONSOLE	Enable the first three configurations CONFIG_CMD_EFICONFIG and CONFIG_CMD_BOOTMENU, U-Boot supports UEFI menu interface. After enabled the configuration CONFIG_AUTOBOOT_MENU_SHOW, UEFI menu can be shown up automatically. To only disply UEFI menu and disable U-Boot console, we can enable the configuration CONFIG_BOOTMENU_DISABLE_UBOOT_CONSOLE, in this case, we also need to remove configuration CONFIG_PREBOOT so can avoid adding boot manager entry in UEFI menu. Please see the details in the document UBOOT-EFICONFIG.
CONFIG_SILENT_CONSOLE	With configuration CONFIG_SILENT_CONSOLE and append "silent=1" into the U-Boot environment (e.g. append it into the macro CONFIG_EXTRA_ENV_SETTINGS), we can totally mute console for U-Boot.

11.2.4 TPM

11.2.5 Firmware TPM

11.2.6 OCI

11.3 Links

- https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8320B.pdf

BIBLIOGRAPHY

[UEFI] Unified Extensable Firmware Interface Specification v2.9, February 2020, UEFI Forum

[EBBR] Embedded Base Boot Requirements v2.0.0-pre1, January 2021, Arm Limited

[fTPM] Firmware TPM, August 2016, Microsoft

[SWTPM] Software TPM

INDEX