TRS Documentation

Release 0.3.0

Linaro

Apr 17, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

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 intracommunication as well as external communication with remote systems. All aspects of security have to be considered, standards need to be addressed and systems need 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 of time and effort.

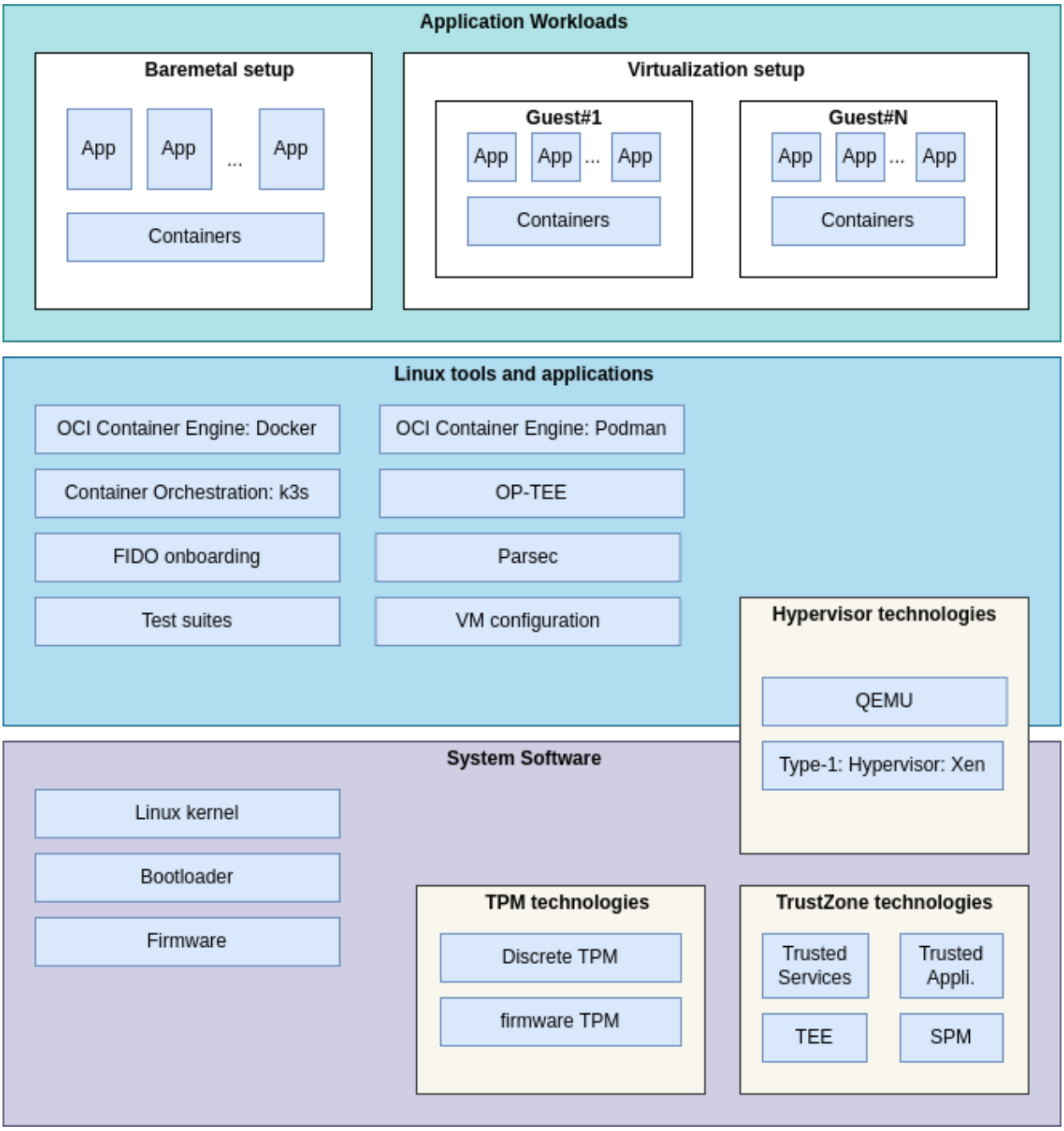
As an answer to the challenges presented, Linaro has created **TRS (Trusted Reference Stack)**, an umbrella project and software stack that includes well tested software. Its components make up a solid base for efficient development and for building unique and differentiating end-to-end use cases.

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.
- An efficient development environment for engineers.
- A product ready reference implementation.
- Configurable to be able to meet different needs.
- Common ground and building blocks for Blueprints and similar targets.
- Interoperability making it possible to use alternative implementations.
- Pre-silicon IP support in environments like QEMU etc.

1.2 High-level overview

The TRS software stack is made up of application software that spans all architectural layers, from low-level programs that communicate directly with hardware to the user-space environment. To make it a flexible solution and be able to partition workloads, it also has support for running applications either as containerized applications or as virtual machines. The image below shows a high level overview of the key components in TRS.


Application Workloads
Baremetal setup
App
App
...
App
Containers
Virtualization setup
Guest#1
App
App
...
App
Containers
Guest#N
App
App
...
App
Containers
Linux tools and applications
OCI Container Engine: Docker
OCI Container Engine: Podman
Container Orchestration: k3s
OP-TEE
FIDO onboarding
Parsec
Test suites
VM configuration
Hypervisor technologies
QEMU
Type-1: Hypervisor: Xen
System Software
Linux kernel
Bootloader
Firmware
TPM technologies
Discrete TPM
firmware TPM
TrustZone technologies
Trusted Services
Trusted Appli.
TEE
SPM

1.3 Firmware software components

Firmware in TRS is provided by Trusted Substrate, which has its own dedicated area in this documentation (see *Firmware - Trusted Substrate*).

1.4 Use-cases and features overview

- Deployment of application workloads via Docker, Podman and k3s.
- Orchestration of resource-managed and isolated application workloads via Docker, K3s, and the Xen type-1 hypervisor providing hardware virtualization.
- Remote updates for all software components, to be able to address security issues, fix bugs and add features. Includes optional A/B updates.
- Measured boot to facilitating sealed keys technologies.
- Encrypted filesystem to meet confidentiality needs.
- Remote attestation to be able to trust and be trusted by remote parties.
- Using backend devices via VirtIO interfaces.
- Early development of new architectural features.
- Ability to load different OSes and distros without hard coded dependencies to firmware (EBBR, SystemReady).
- General Arm ecosystem enablement.
- Regression testing on merge requests and on daily builds.

1.5 TRS system architectures

TRS is compatible with both bare-metal and virtualized environments. This allows TRS to be utilized for a variety of application cases. When needed on typical embedded devices, baremetal is likely the best option; but, if you need to run multiple payloads, operating systems, etc., virtualization architecture may be a better fit. In the section *System Architectures*, you will find a detailed overview of the two different architectures.

1.6 Feedback and support

To request support please contact Linaro at support@linaro.org.

1.7 Maintainer(s)

- Joakim Bech <joakim.bech@linaro.org>
- Ilias Apalodimas <ilias.apalodimas@linaro.org>
- Mikko Rapeli <mikko.rapeli@linaro.org>

CHAPTER

TWO

USER GUIDE

2.1 Initial setup

To be able to build TRS, we must take a few steps to prepare our host environment. Typically, this involves installing a few essential packages and initializing scripts to setup paths etc. Following the steps listed here should result in a host environment capable of building, deploying, and running software on an emulated environment such as QEMU or on one of the TRS-supported devices.

It is important to note that it is difficult to provide completely accurate instructions since there are so many Linux distributions, each with subtle variations. They have distinct package managers, various package naming conventions and different package versions. So, our instructions have been mostly tested with Ubuntu LTS and, now, with Ubuntu 22.04 LTS.

2.1.1 repo

For more than a decade now, `repo` has been the Android tool responsible for checking out groups of individual git repositories that make up a larger project or product. At its core, repo reads manifest files, which are xml-files that include URLs to remote servers hosting gits. In the xml-files, we also find the names of the gits used in a project, as well as the branch or commit we're tracking.

repo is very configurable, but we use it mostly to obtain the source code for our development builds and to generate stable releases. Repo's excellent method of leveraging local mirrors, which considerably reduces setup time and saves a substantial amount of disk space, was a major factor in the decision to use it. How this is accomplished is described in depth later in this paper.

Install repo

Notice that here you do not install a massive SDK; instead, you just download a Python script and place it in your `$PATH` variable. See the Google repo sites for specific instructions on "installing" repo before proceeding.

2.1.2 Getting the source code

This step gets the code necessary to build TRS. You may either checkout a version tracking the latest on all gits (a "developer setup"), or you can checkout a specific release. The difference is shown in the highlighted `repo init` lines in the examples below, where you provide different branch names and manifest files.

Developer setup

```
$ 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
```

Release build

```
$ 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
```

2.1.3 Getting the host packages

As previously explained, various host packages are needed for building TRS. These packages are a combination of distribution packages and a few required Python packages. The distribution packages will be installed with the host's package manager, while the Python packages will be installed with pip. The latter are only required to build the documentation.

Distribution packages

Your **sudo** password will be required to complete the steps listed here.

Ubuntu 22.04 LTS

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

This will install the following packages:

```
acpica-tools
adb
autoconf
automake
bc
bison
build-essential
ccache
chrpath
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
zstd
```

The package list comes from Yocto documentation. Make sure to check the documentation if a build error is produced because of a missing package.

Python packages

Note: Python packages are not required to build TRS images. Follow this step only if you plan to change and test TRS documentation.

All Python packages are installed by default in `<workspace root>/.pyvenv` using a virtual Python environment. The advantage of doing so is that there will be no traces of the Python packages required for TRS if we delete the `.pyvenv` folder. This approach may eventually avoid conflicts with other tools that require other versions of some Python packages.

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

2.1.4 Initial sourcing

Newly installed Python packages have to be made available to the virtual Python environment by sourcing the "activate" script.

Note: Here, you only need to do the "sourcing" step once per shell where you want to start the build. This means that if you **forget** to run this after spawning a new shell, your build will likely fail.

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

Start the build

Next, we start the build, which will likely take a few hours on a standard desktop machine the first time you build it with no caches primed. The TRS is based on multiple Yocto layers and if you don't already have the environment variables `DL_DIR` and `SSTATE_DIR` set, they will be set to `$HOME/yocto_cache` by default. Note that, `make clean` does not clear the download and sstate caches and therefore doesn't affect the build time negatively. The actual build is started by the following:

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

After following the steps above, please continue with the target specific instructions, which you will find in the navigation menu to the left.

2.1.5 Tips and tricks

Reference local mirrors

As the project grows, the time required to do the initial `repo sync` increases. The `repo` tool can reference a locally cloned forest and clone the majority of the code from there, taking just the eventual delta between local mirrors and upstream trees. To do this, add the argument `--reference` to the `repo init` command, for instance:

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

Use local manifests

In some cases we might want to use another remote, pick a certain commit or even 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.

Covers the steps necessary to configure your host system to develop and deploy TRS onto supported devices.

2.2 Targets

2.2.1 QEMU setup

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

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.

Hint: 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:~$
```

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_SUITE` 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.

2.2.2 Baremetal

This section will explain how to run TRS on hardware. There are primarily two steps that must be taken. We should a) flash the device firmware to a suitable medium and b) prepare a USB disk with the operating system.

Flashing the firmware

The firmware of a device is unique to that device. As a result, each supported device has its own set of tools and requirements to follow when writing the firmware. Device specific flashing instructions for TRS supported devices can be found at the *Installing firmware* page.

> **Warning:** If the firmware is to be flashed on an SD card, the SD card should be found under `/dev/sdX`, where X is a letter associated to the card as recognized by the host system (ex. `/dev/sda` or `/dev/sdb`).

Flashing the OS and rootfs

TRS builds an OS and a root filesystem that are compatible with all devices, as opposed to the device firmware that is device-specific. Therefore, it is sufficient to write a "wic-file" to a device (eMMC or SD card) in order to get the OS image and root filesystem onto TRS supported devices. In TRS, it's the image named as "trs-image-trs-qemuarm64.wic", that should be used. Although the wic-file name contains QEMU, it is the correct one and should be used for all devices.

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

Boot TRS

Plug-in the USB stick, SD-cards and reset the device. If the USB stick or SD-cards is detected, TRS will boot automatically.

Hint: Prefer USB 3.0+ ports always. If you are experiencing difficulties booting TRS, interrupt the U-Boot boot sequence and verify that your disk is identified.

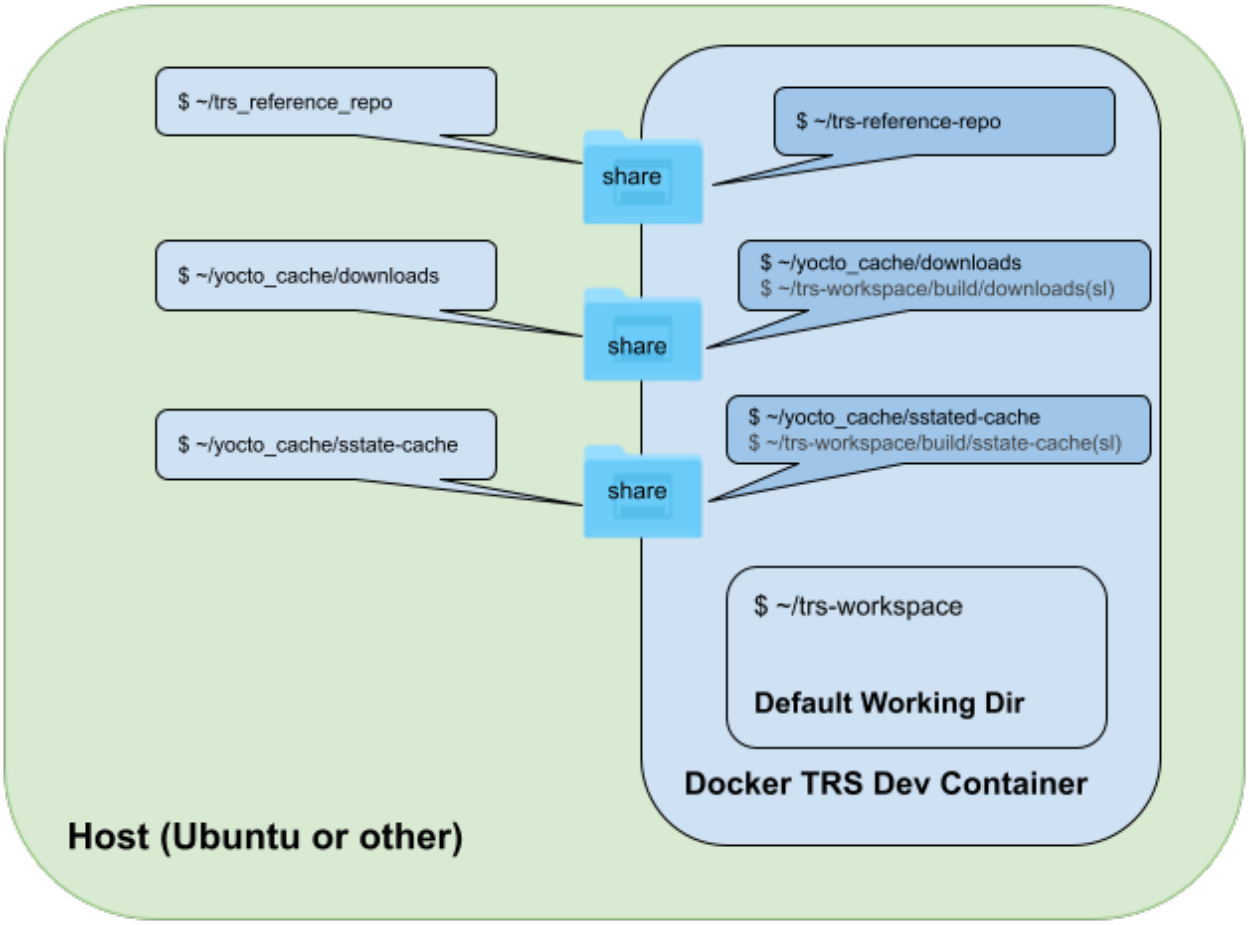
```
=> usb start
=> usb storage
  Device 0: Vendor: SanDisk Rev: 1.00 Prod: Cruzer Blade
            Type: Removable Hard Disk
            Capacity: 29340.0 MB = 28.6 GB (60088320 x 512)
```

2.2.3 Docker

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.

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.

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.

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`.

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.

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
```

2. Build Docker Image (Host)

Create a docker image, 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
```

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. 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

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
```

(continues on next page)

(continued from previous page)

```
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!

5. 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 *QEMU setup* section of this document.

Describes how to work with various devices.

2.3 Extend

TBD

Describes how TRS may be customized and run on unsupported target platforms.

CHAPTER

THREE

DEVELOPER MANUAL

3.1 System Architectures

From a high level, TRS provides two main variants, the baremetal variant and the virtualization variant. Both are built from the same sources with minor modifications.

Baremetal and virtualization are two independent software deployment strategies for computer systems. Baremetal refers to the installation and running of an operating system without any intervening software layers. In other words, the OS directly interacts with the machine's physical hardware, including the CPU, memory, storage, and networking devices etc.

Virtualization, on the other hand, involves the creation of one or more virtual machines (VMs) that run on top of a hypervisor, which is a software layer that abstracts the underlying physical hardware and exposes it to the VMs as a set of virtual hardware components. Each VM have the ability to run its own operating system, which interacts with the hypervisor's virtual hardware. This essentially means that it is possible to run several different operating systems concurrently on the same physical hardware.

So, baremetal systems communicate directly with the real hardware, whereas virtualized systems communicate with the hypervisor's virtual hardware. Ultimately, the decision between bare metal and virtualization is determined by the individual use case and system requirements. Virtualization is often used in data centers and other business contexts where flexibility, administration, and effective resource utilization are essential. But there is a trend where other markets started looking at running virtualized environments. Automotive is one, but we've even seen that mobile handset have use cases where they want to leverage virtual environments for various workloads.

Since TRS have the ability act as both baremetal and a virtualized architecture, with and without using container technologies, we believe that TRS is well suited for a lot of use cases and scenarios.

3.1.1 Baremetal architecture

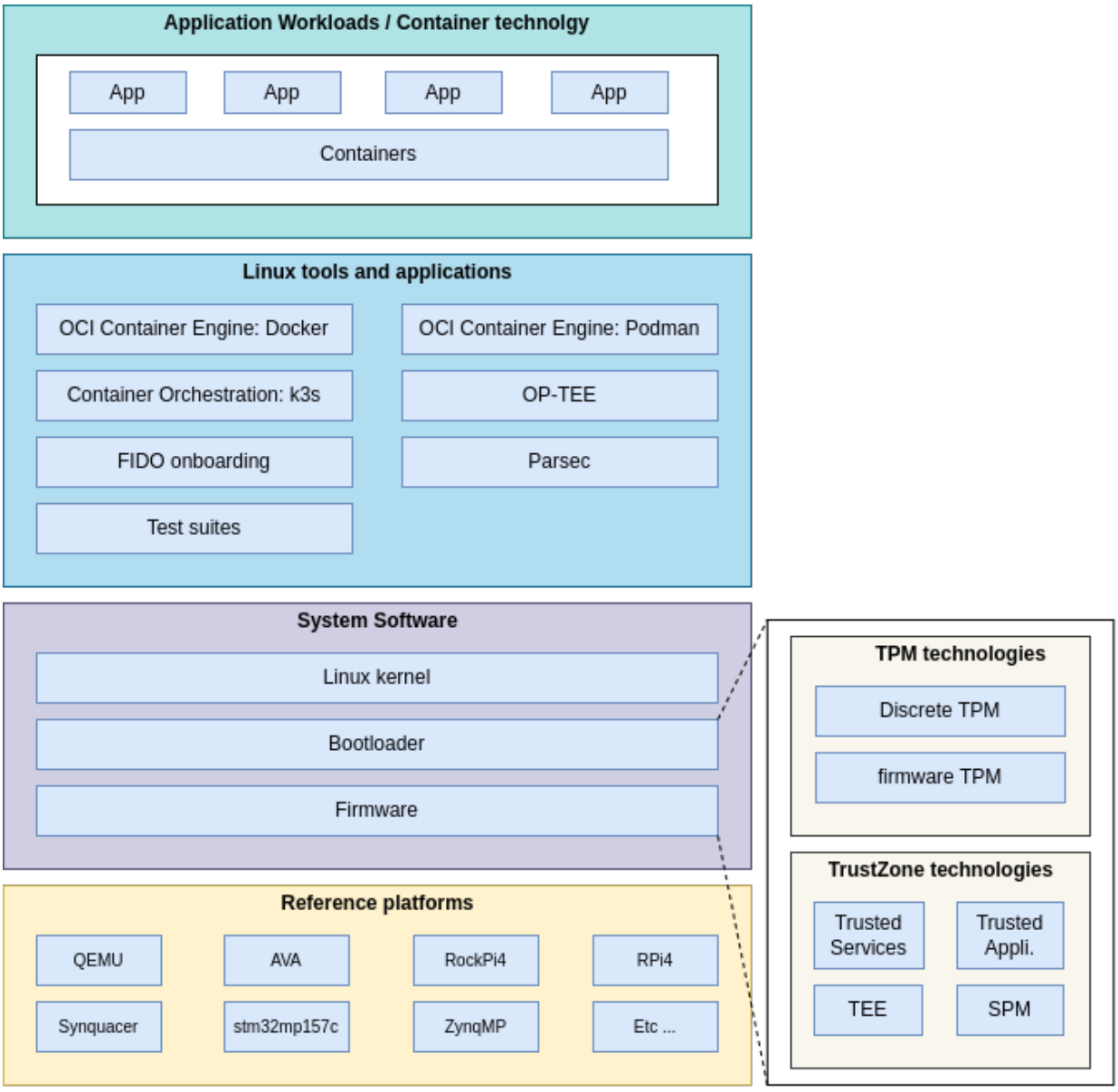
Conceptually, TRS has code running in every architectural layer. With an Armv8-A device, this means that code will run on all exception levels on both the non-secure and secure (TrustZone) side. The diagram below does not show a typical image with secure and non-secure sides and exception levels; rather, it is intended to provide a quick overview of the kinds of building blocks we are likely to see being used in an TRS baremetal setup.

At the very top, we have (OCI) containerized workloads. Linaro will be able to provide workload applications designed to solve specific use cases; these are often included in Arm and Linaro Blueprint offerings. Additional container images might be directly retrieved from image registries.

The next layer consists of standard Linux tools and apps, which are often user-space applications or libraries that provide and abstract diverse hardware. Parsec, for example, is intended to abstract security hardware such as TEE environments, TPMs, and HSMs, among others. On the security side, we also find libraries that enable direct communication with OP-TEE. This conforms to the GlobalPlatform Client and Internal API standards. This layer also contains the numerous test tools and test suites required to guarantee API correctness, device stability and robustness.

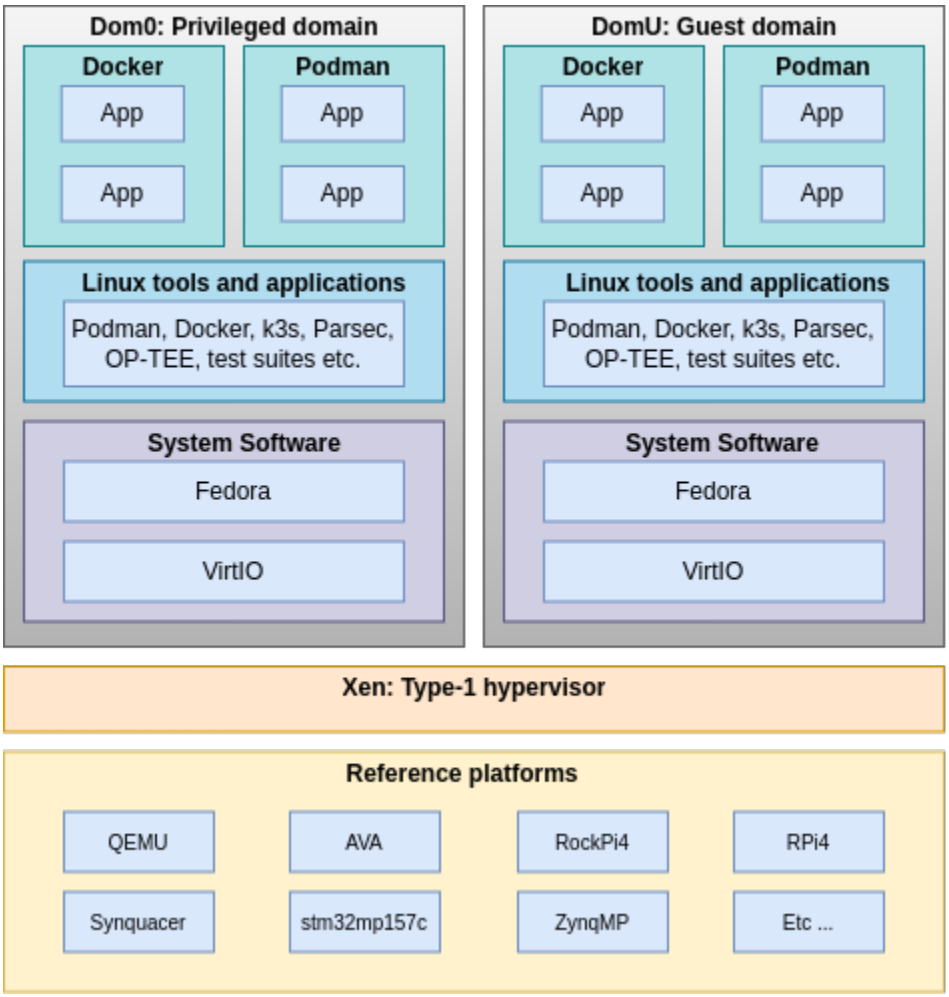
The last software layer consists of the Linux operating system and low-level hardware. We've also included the bootloader, which is mostly utilized during boot. Yet, even bootloader code has portions that remain resident after the main operating system gains control. For example, the secure monitor code (BL31) never stops running and the same is true for the OP-TEE OS code, which provides security features to the main OS for various use cases.

Then at the bottom we've included a hardware layer showing the type of devices that we support in TRS. This is something that will change over time. For an up-to-date list, please have a look at our CI matrix that is continuously updated and shows all the devices that we test on a regular basis.



3.1.2 Virtualization architecure

As for the virtualization architecture, TRS supports hardware virtualization by using Xen, which is a well-known type-1 hypervisor implementation. If we look at the block diagram below, we mostly find the same bits and pieces as we saw in the block diagram for the baremetal architecture. The major difference is the hypervisor layer that has been included between the hardware and the rest of the software. The hypervisor layer is responsible for managing the virtual machines and allocating resources to them. It allows multiple operating systems to run on the same physical hardware simultaneously. As shown in the block diagram above, you can also use container applications to run separate workloads if guest separation isn't enough.



3.2 User Accounts

TRS leverage the EWAOL distribution user configuration, which has defined the following user accounts:

- `root` with administrative privileges enabled by default. The login is disabled if `ewaol-security` is included in `DISTRO_FEATURES`.
- `ewaol` with administrative privileges enabled with `sudo`.
- `user` without administrative privileges.

- `test` with administrative privileges enabled with `sudo`. This account is created only if `ewaol-test` is included in `DISTRO_FEATURES`.

By default, each users account has disabled password. The default administrative group name is `sudo`. Other sudoers configuration is included in `meta-ewaol-distro/recipes-extended/sudo/files/ewaol_admin_group.in`. For virtualization images, above user accounts are created for Control VM and Guest VM domains.

If `ewaol-security` is included in `DISTRO_FEATURES`, each user is prompted to a set new password on first login. For more information about security see Security Hardening.

All `validation_run-time_integration_tests` are executed as the `test` user.

3.3 Build System

Yocto with bitbake is the build engine used to build TRS. However, due to the fact that a TRS builds various components such as firmware (Trusted Substrate), root-filesystem, and kernel independently, we have wrapped the setup, configuration, and build commands in a simple high-level Makefile. That way it easy for people to get started with TRS by minimizing the number of commands needed to get it working. Interested parties can also look at the inner working of the Makefile to better understand how things come together.

Putting it in a Makefile makes it easy to additionally address any build dependencies, i.e., we can control the build sequence, which may be desirable and needed for an umbrella project such as TRS.

3.3.1 Target Platforms

TRS supports multiple platforms, including Arm application profiles, execution states, and extension types. However, TRS use the same OS and root filesystem image for all devices. Therefore, there is currently not a need to describe the platforms as such. Refer to the *Firmware - Trusted Substrate* section to learn how to configure and build firmware for different target systems.

3.3.2 Distribution Image Features

TRS distro features can be found under `./trs/meta-trs/conf/distro/trs.conf`. In summary we append the following `DISTRO_FEATURES` in TRS.

```
grep: ../../meta-trs/conf/distro/trs.conf: No such file or directory
```

As can be seen, we inherit and use EWAOL's `ewaol-baremetal` build for our baremetal environment. Likewise for our **virtualization** build, we get Xen by enabling `ewaol-virtualization` that can be found in `./meta-ewaol/meta-ewaol-distro/conf/distro/ewaol.conf`

3.4 Yocto Layers

TBD

3.4.1 TRS recipes

TRS leverage various layers and recipes to build firmware, root filesystem and various images. The best place to start looking for those recipes is in the default manifest file located in trs-manifest.git repository.

3.5 Security

3.5.1 Hardening Flags

By enabling the security features established by Project Cassini, TRS distribution images can be hardened to limit potential sources or vectors of security vulnerabilities. Currently this is achieved by enabling a couple of different `DISTRO_FEATURES`. to:

- Force password update for each user account after first logging in. An empty and expired password is set for each user account by default.
- Enhance the kernel security, kernel configuration is extended with the `security.scc` in `KERNEL_FEATURES`.
- Enable the 'Secure Computing Mode' (seccomp) Linux kernel feature by appending `seccomp` to `DISTRO_FEATURES`.
- Ensure that all available packages from `meta-openembedded`, `meta-virtualization` and `poky` layers are configured with: `--with-libcap[-ng]`.
- Remove `debug-tweaks` from `IMAGE_FEATURES`.
- Disable all login access to the `root` account.
- Sets the umask to `0027` (which translates permissions as `640` for files and `750` for directories).

3.5.2 Threat models

Note: This threat model section will be reworked and some of the information in here will be moved over to the firmware section.

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.

Use cases

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`

Other projects threat models

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.

OP-TEE threat model

TBD

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.

U-Boot threat model

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.

TPM threat model

TBD

Firmware TPM threat model

TBD

OCI

TBD

3.5.3 Links

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

3.6 Features

The purpose of this page is to provide an overview of the essential features of the Trusted Reference Stack (TRS), including it communication capabilities with internal and external system components. This information aims to help readers understand what TRS can offer in terms of its functionality and interoperability.

3.6.1 TRS features

Secure Boot

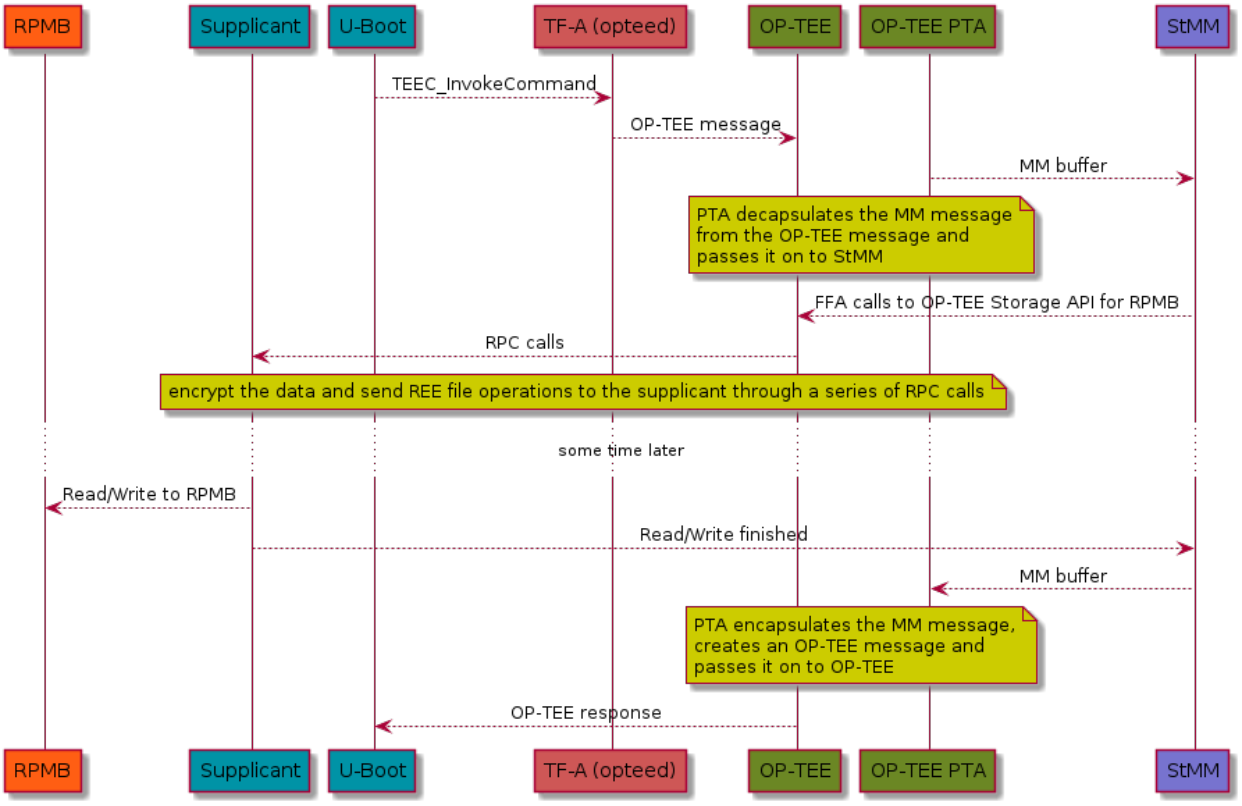
The firmware component of TRS unconditionally enables UEFI secure boot for all supported platforms. There are some hardware requirements that will dictate how Secure Boot is configured and enabled on your hardware. [UEFI] (section 32.3.6 Platform Firmware Key Storage Requirements) specifies that the Platform (PK) and Key Exchange Keys (KEK) must be stored in tamper-resistant nonvolatile storage. On Arm servers this is usually tackled by having a dedicated flash which is only accessible by the secure world. Below is a table describing the security features enabled by various hardware entities.

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

In the embedded case, we typically don't have a dedicated flash. What's becoming more common though is eMMC storage devices with an RPMB partition. The eMMC storage devices are solid-state storage devices that leverage flash memory technology to provide affordable and reliable storage for small electronic devices. The eMMC device's RPMB partition (Replay Protected Memory Block) provides a secure storage area for sensitive data using a replay protection mechanism to prevent unauthorized access and modification. Because of this, the eMMC devices has become a key component for numerous electronic devices, serving as dependable and secure data storage. Trusted Substrate will use that RPMB partition to store all the EFI variables, if the device runs OP-TEE and have a RPMB partition.

For devices without an RPMB, the UEFI public keys (PK, KEK, DB, etc.) will be embedded in the firmware binary. The wrapping of these keys has its own limitations and consequences. You have to make sure that the public keys are immutable, something that is typically done by tying them to the Root of Trust (ROT). To update any security-related EFI variables, you must update the firmware. By default, you can only run binaries that have been digitally signed. Other EFI variables that are not security-critical are stored in a file within the ESP.

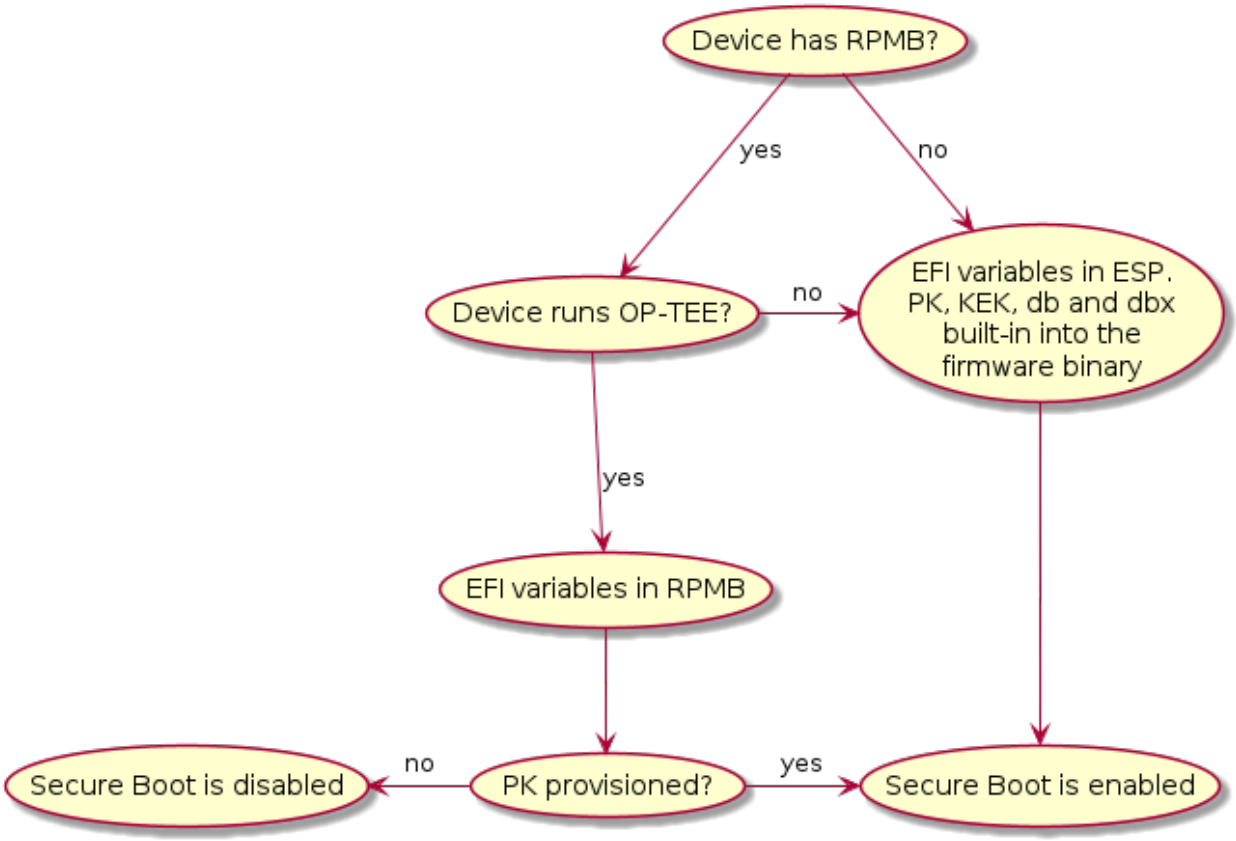
On the sequence diagram below, we see a typical scenario, describing the different components involved when storing and retrieving data from a RPMB partition.



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

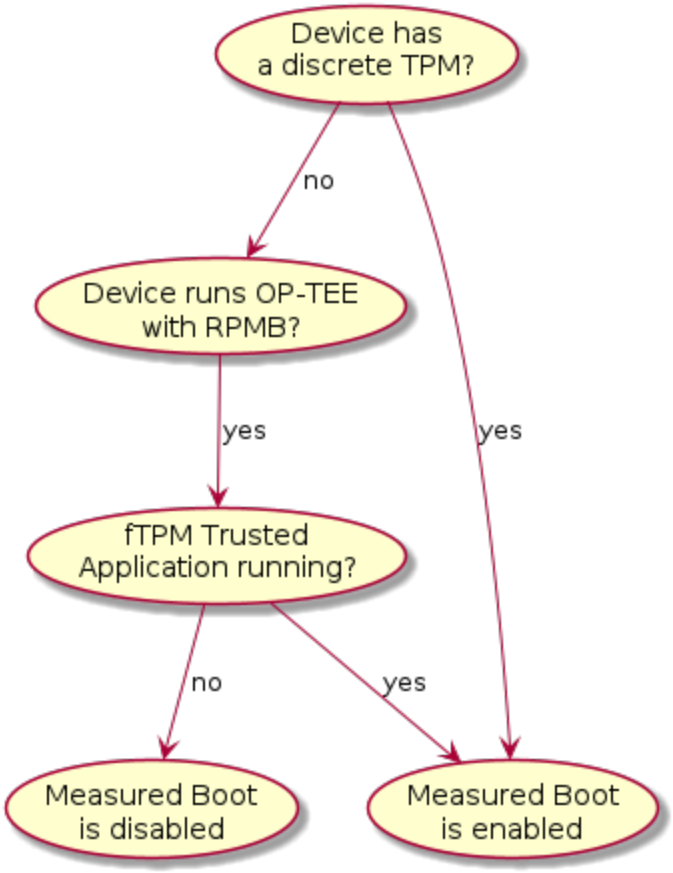
Secure boot limitations

The firmware automatically enables and disables UEFI Secure Boot based on the existence of the Platform Key (PK). As a consequence, devices that embed keys into 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, devices that stores the variables in the RPMB come with an uninitialized PK. As such the user must provide a PK during the setup process in order to enable Secure Boot. The diagram below illustrates how a device can be set up to have secure boot enabled or disabled.



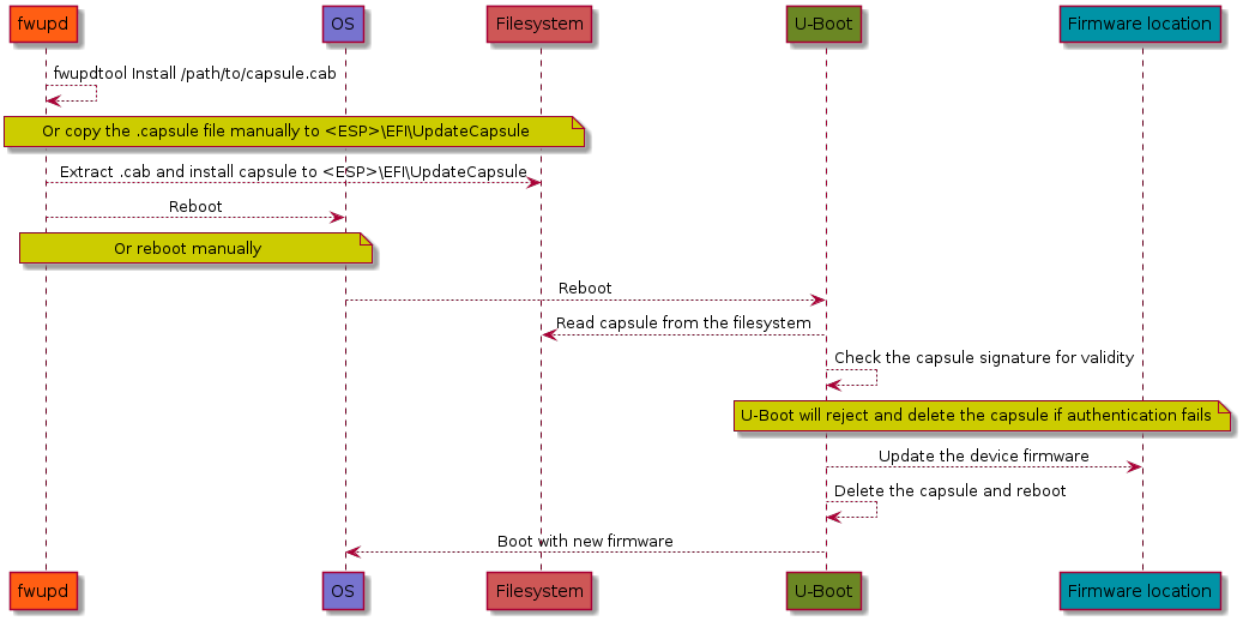
Measured Boot

TRS has been designed to take advantage of TPM devices. 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. During the first OS boot, it will automatically look for a TPM device. If such a TPM device is present it will generate a random key, encrypt the root filesystem and seal it against measurements found in *PCR7* which holds the Secure Boot Policy and EFI keys used for UEFI Secure Boot. Trusted Substrate supports discrete TPMs as well as firmware based TPMs. Which one being used depends on the device capabilities and the software available. The diagram below illustrates how a device ends up running with measured boot enabled or disabled.


Device has
a discrete TPM?
no
yes
Device runs OP-TEE
with RPMB?
yes
fTPM Trusted
Application running?
no
yes
Measured Boot
is disabled
Measured Boot
is enabled

Authenticated Capsule Updates

TRS can update the device firmware using Authenticated capsule updates on-disk A more detailed explanation is included in our *Updating the firmware* chapter, but the sequence diagram that follows should provide enough information on how the firmware is updated.



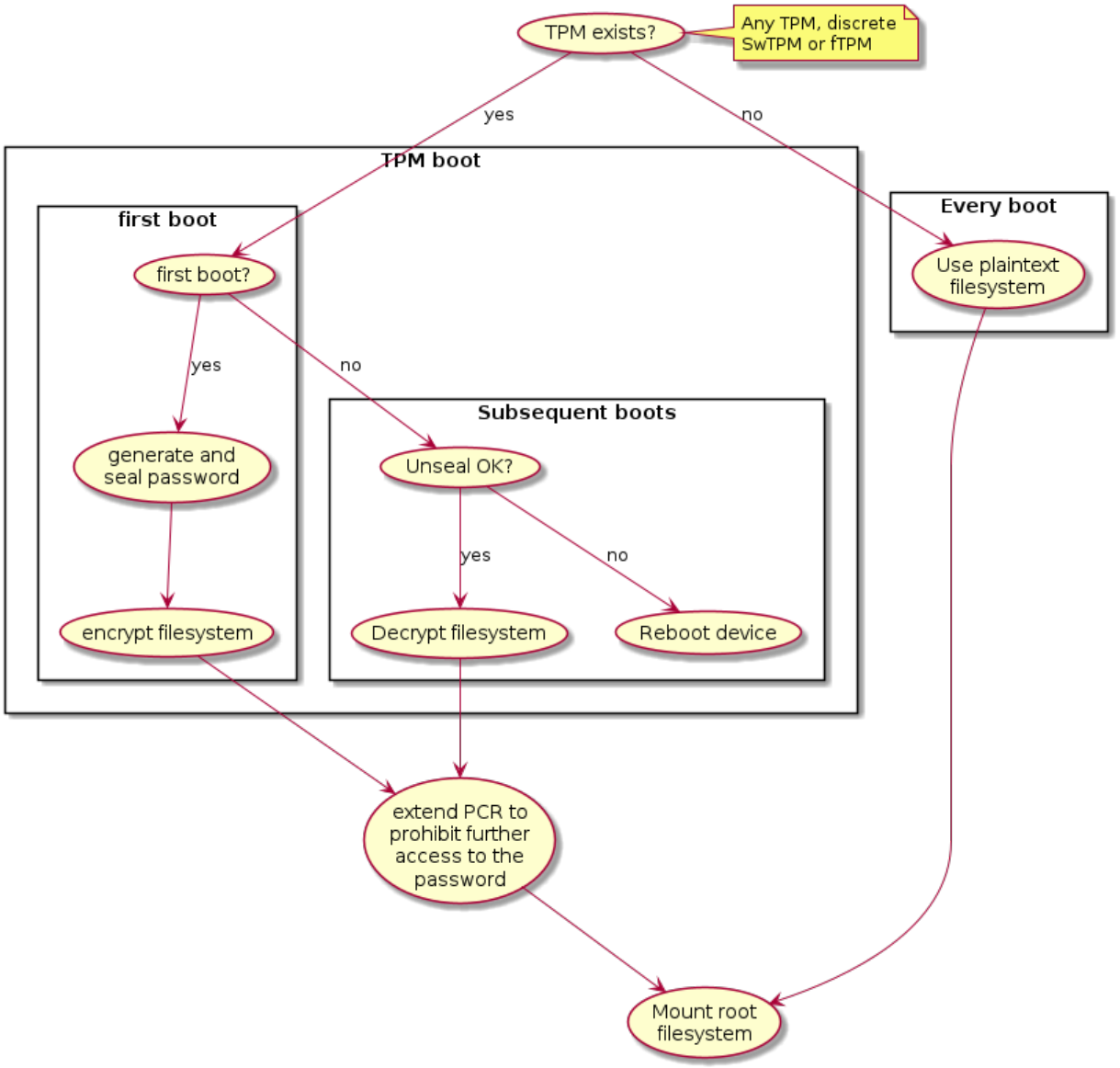
Disk encryption

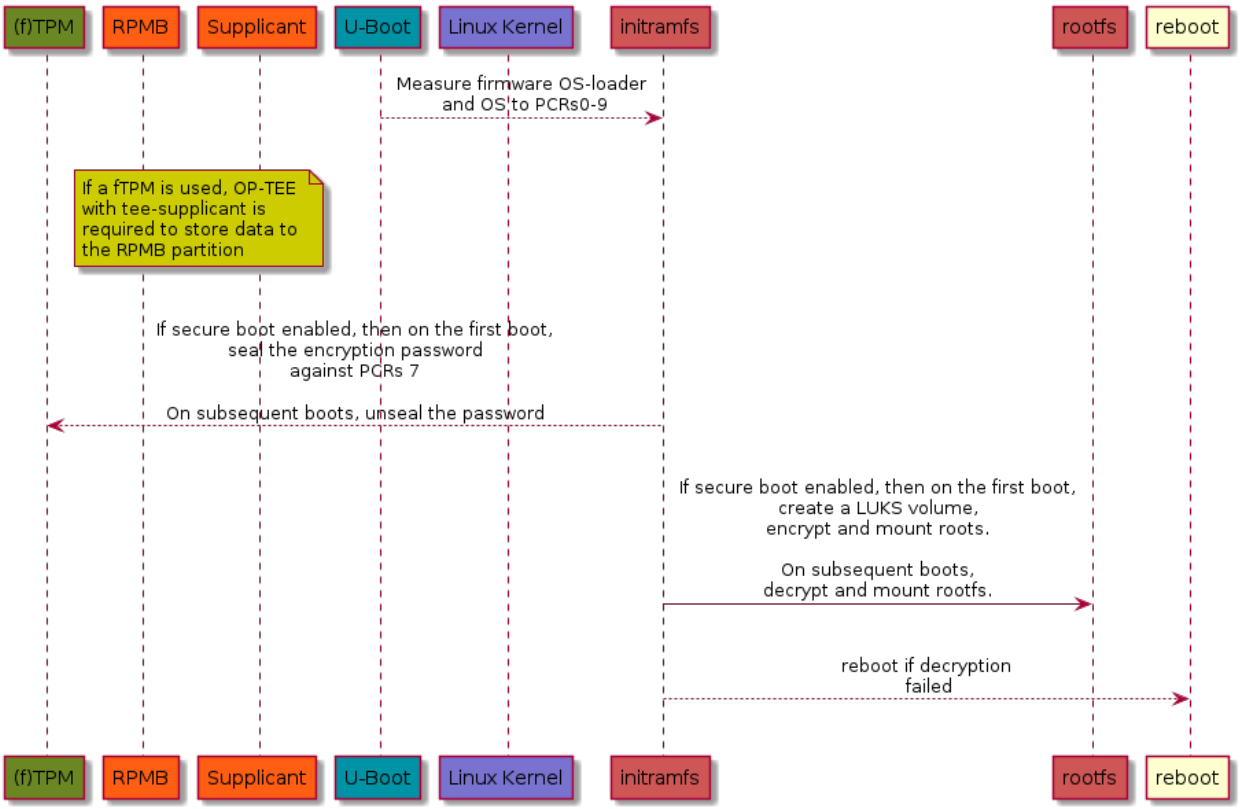
The TRS build is by default configured to look for a TPM device. If it does, it will generate a random password during the first boot, seal (see *TPM sealing*) it against PCR7 which is the PCR meant to be used for the secure boot state. Using this password, it will encrypt your root filesystem using `aes-xts-plain` algorithm and block mode. This is something that will happen regardless of TPM implementation. If on the other hand there is no TPM available, then the devices will use a plaintext/unencrypted filesystem. This is all explained in the graph below, where you can follow the steps from booting up the device to the filesystem being mounted.

Important:

- The encryption here will start from **U-Boot**, i.e, in TRS it's currently U-Boot that is the first component leveraging the TPM device. This is something that might change in the future.
- It's also worth to pay attention to the step that extends the PCR once more to prohibit access to the password.

To get an idea of the components involved, please have a look at the sequence diagram below showing the call flow between the components involved when setting up disk encryption in TRS using a TPM device.


TPM exists?
Any TPM, discrete SwTPM or fTPM
yes
no
TPM boot
first boot
first boot?
yes
no
generate and seal password
encrypt filesystem
Subsequent boots
Unseal OK?
yes
no
Decrypt filesystem
Reboot device
Every boot
Use plaintext filesystem
extend PCR to prohibit further access to the password
Mount root filesystem



Virtualization

So far, TRS uses Xen as the Hypervisor for Virtualization use cases. When Xen is enabled, the GRUB menu provides an entry TRS `Xen (if supported)` making it possible to boot the Xen hypervisor. What you will see is something similar to the image below.

Xen hypervisors' EFI program and configuration file (`xen.cfg`) both are located in the root folder of boot partition. The configuration file contains settings for Xens' log levels when it comes debugging, it also contains the path to the Linux kernel image, the Linux kernel command line, etc. The Xen hypervisor parses the configuration file and boots Linux kernel image.

Note, the Xen hypervisor doesn't load initial ramdisk, this is different from the boot flow in bare metal mode which loads both the initial ramdisk as well as the 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 a booting up the system successfully, we can use the command `xl list` to list Xen domains. The Xen `Dom0` with naming `Domain-0` is created by default and it will look like this:

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

The goal of TRS is to support both `Dom0` and `DomU` with the same root filesystem image. However, if Xen `Dom0` automatically boot up Xen `DomU` from the root filesystem, Xen `DomU` will automatically boot the next level's Xen `DomU`, and so on, causing a nesting issue. For this reason, the TRS root file system does not contain anything for Xen `DomU`. To deploy a virtual machine for Xen `DomU`, the procedures outlined below must be followed. 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/` in order for the virtual machine to be launched automatically upon subsequent booting. After that, 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
```

Now we 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*. Now we must copy the kernel image. The virtual machine can use the kernel image already stored in `/boot/Image` for the Xen `Dom0`. With the previous preparations, Xen `DomU` is prepared to run the virtual machine. With the command shown here, we can create the VM:

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

After the VM has been created, 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). To access Xen's `DomU` console, you can use the command `xl console` followed by a domain name, as exemplified here:

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

To leave the DomU console and return to `Dom0`, you can press `ctrl-[`.

Known Xen issues with TRS

1. **Platforms**: Currently Xen hypervisor is only supported for ADLink AVA platform.
2. **Images**: The Xen hypervisor loads kernel image but it doesn't load initial ramdisk.
3. **TPM support**: `Dom0` currently does not support TPM. 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 will not be possible to reuse the root file system image due to Xen not supporting TPM at the current stage.

3.6.2 Technologies and software

This section intends to give a high level overview of the key technologies and software that is used in TRS. It is meant to be an introduction rather than an full description.

TPM - Trusted Platform Module

A TPM (Trusted Platform Module) device is a hardware-based security device that offers cryptographic operations, secure storage, disc encryption and attestation services. Its main objective is to ensure the integrity of key system components and secure sensitive data from unauthorized access in order to establish a secure foundation for a computing system. A unique feature that TPM devices offer is the so called Platform Configuration Registers (PCRs), which are used to measure the system configuration and software. PCRs start zeroed out and can only be reset with a system reboot. PCR's can be extended by writing an appending digest (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 )
```

TRS supports three different configurations, that is a real TPM hardware chip, [fTPM] or [SWTPM] if using QEMU. On an API level, they're all equivalent, but the security and performance implications are different. The fTPM solution is flexible in the sense that it runs as Trusted Application, so it's easy to change and update it if needed. From performance point of view, it's faster that a real TPM chip, since it's running on a fully fledged Cortex-A core. However, to be able to use the fTPM, the system must have reached a state where OP-TEE is up and running, since that is where the code is running. The other software based solution, SWTPM, is a piece of software that is started as a separate binary and exposes itself via sockets. That makes it possible to use a TPM device already from the first boot loader (if drivers exist!). Exactly how that work can be found in the TPM and U-Boot blog post. A real, discrete TPM chip will of course also be available directly from the boot. How to hand over the ownership of the TPM between different execution environment and to ensure that there are drivers capable of communicating with the TPM device is a technical challenge shared between all setups. Another issue that needs to be addresses is TPM impersonator, man-in-the-middle attacks. Something that real TPM devices connected with I2C and SPI are susceptible to, see for example the TPM Genie attack.

TPM sealing

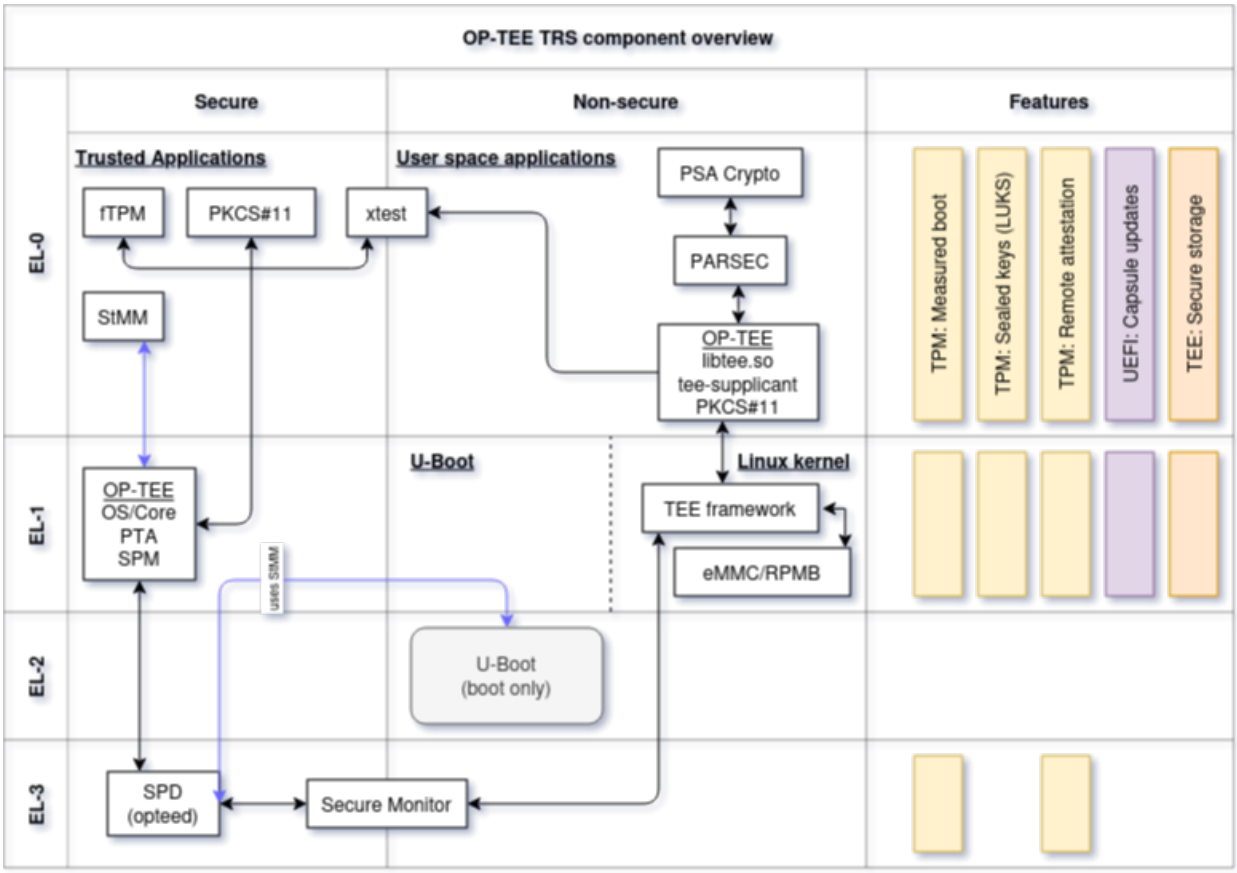
TPM **sealing** is a technique that allows locking keys and data to a certain PCR state. In other words, when we say that we "seal a key," what we actually mean is that after a certain number of PCR measurements, we take a key of our choice and ask the TPM to store and lock it to that specific PCR state. The only way to unlock the key once that has been completed is to ensure that we obtain the same PCR in subsequent boots. Since the PCR measurements are based on collision resistant cryptographic algorithms, it's extremely unlikely to be able to brute-force this schema. To this date, SHA-256 and higher isn't susceptible to brute-force attacks. Weaker algorithms such as MD5 and SHA-1 on the other hand are no longer considered secure and shall not be used any longer. The concepts described here with locking keys to PCR's is the bare bone when it comes to encrypting keys and other secrets during boot etc.

OP-TEE

OP-TEE is an open source TrustZone solution, a so called Trusted Execution Environment (TEE) that makes it possible to run code and keep sensitive data away from the normal OS environment. The OP-TEE solution is made up of code running in a both secure and non-secure contexts. The secure side, is where the main OP-TEE OS runs (at `S-EL1`) together with the Trusted Applications (at `S-EL0`). On the non-secure side OP-TEE has a TEE driver that rely on the TEE framework provided by Linux kernel. To support clients (normal Linux applications), OP-TEE also provides a couple of libraries, giving API access to TEE communication and features (`libteec.so`, `tee-supplicant` and a few others). TRS uses OP-TEE for a number of reasons with the most notable ones being:

- Implement and run [fTPM] if the hardware doesn't have a discrete TPM.
- Store EFI variables when the device has a RPMB partition.
- Provide a Deterministic Random Bit Generator (DRBG) if the hardware doesn't provide a True Random Number Generator (TRNG).
- Implement a PKCS#11 backend 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.

LUKS - Linux Unified Key Setup

Block devices, like filesystems and swap partitions, can be encrypted using the disk encryption system called LUKS. Conceptually, LUKS protects the data by leverage keyslots. Keyslots may include several kinds of keys, such as passphrases, OpenPGP public keys, or X.509 certificates. Encryption is carried out using a multi-layer technique. There are two versions of LUKS, with LUKS2 providing additional capabilities such robustness to header corruption and default use of the Argon2 encryption algorithm.

Xen

Xen is an open-source type-1 or baremetal hypervisor that allows multiple instances of the same or different operating systems to run on a single machine. It is used in various applications targeting different environments, including server and desktop and embedded. The Xen Project hypervisor has a small memory footprint, is independent of operating systems it is running, it isolates drivers and it also supports paravirtualization, a technique, that allows multiple operating systems to share system resources more efficiently. Paravirtualization improves performance and reduces overhead by enabling direct communication between the guest operating system and the hypervisor. Xen manages CPU, memory, and interrupts while running directly on the hardware. The VM's runs on top of the hypervisor. A specialized and more privileged VM, called `Dom0`, comprises system services, device drivers and software to manage a Xen-based system. Alongside with that, there are usually other guests running as VM's as well, we refer to those as DomU. For more details about the Xen project, please have a lot at the Xen Project.

CHAPTER

FOUR

FIRMWARE - TRUSTED SUBSTRATE

4.1 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.

4.2 Hardware and Software

4.2.1 Supported Platforms

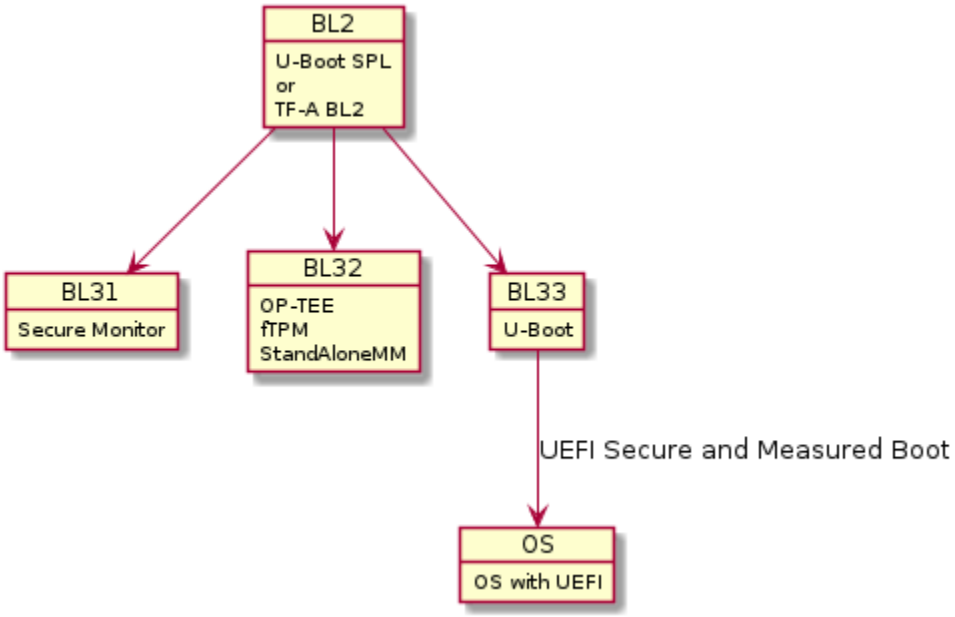
Trusted Substrate supports a variety of armv8 and armv7 boards. It's important to understand that the hardware characteristics dictate the supported features as well as the level of the device security

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

4.3 Build and install

4.3.1 Getting the firmware

Building from source

Trusted Substrate is part of TRS, same steps apply to build from source as documented at *Initial setup*.

Building meta-ts from source

By default *make meta-ts* will build firmware for all supported boards.

> **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
>
> *Secure boot limitations* for hardware limitations

Compiling for different boards is straightforward. To build only one board firmware, pass *TARGET* to *make* with board name.

```
make TARGET=<BOARD> meta-ts
```

replace **<BOARD>** with:

- qemuarm64-secureboot

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

The build output is in `build/tmp_<BOARD>/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.

4.3.2 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 0x400000  (Control-A S -> fip.bin-synquacer)
```

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-<ev1|dk2>.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**

4.3.3 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.

4.4 Configuration and OS booting

4.4.1 Configuring UEFI variables

Boards that embed the UEFI keys in the U-Boot binary *Secure boot 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'
```

(continues on next page)

(continued from previous page)

```
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
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
```

4.4.2 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 *Secure boot 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 precompiled firmware binaries you can test that with our LEDGE Reference Platform.

Download TRS

Download a .wic.gz image from here extract and rename it

```
gunzip ledge-iot-ledge-qemuarm64-<date>.rootfs.wic.gz
mv ledge-iot-ledge-qemuarm64-<date>.rootfs.wic ledge-iot.wic
```

Running TRS

Throughout the examples we will be using a USB disk. You can prepare one with

```
cat ledge-iot.wic > /dev/sdX
```

Note: LEDGE RP will automatically encrypt your root filesystems if measured boot is enabled. Since it also enables SELinux by default it will reboot once due to filesystem relabeling. Be patient this only happens on first boot.

Before first boot you need to prepare the firmware EFI variables accordingly. You only need to interrupt the bootloader and issue the `efidebug` commands once.

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  dir:///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
```

```
qemu-system-aarch64 -m 2048 -smp 2 -nographic -cpu cortex-a57 \
    -bios flash.bin -machine virt,secure=on \
    -drive id=os,if=none,file=ledge-iot.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 efi/boot/bootaa64.efi -i virtio 0:1 ledge-
↪initramfs.rootfs.cpio.gz -s 'console=ttyAMA0,115200 console=tty0 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 efi/boot/bootaa64.efi -i usb 0:1 ledge-initramfs.
↪rootfs.cpio.gz -s 'console=ttyAMA0,115200 console=tty0 root=UUID=6091b3a4-ce08-3020-
↪93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

Run on stm32mp157c dk2 or ev1

```
=> efidebug boot add -b 1 TRS usb 0:1 efi/boot/bootarm.efi -i usb 0:1 ledge-initramfs.
↪rootfs.cpio.gz -s 'console=ttySTM0,115200 console=tty0 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 efi/boot/bootaa64.efi -i usb 0:1 ledge-initramfs.
↪rootfs.cpio.gz -s 'console=ttyS2,1500000 console=tty0 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 efi/boot/bootaa64.efi -i usb 0:1 ledge-initramfs.
↪rootfs.cpio.gz -s 'console=ttyAMA0,115200 console=tty0 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 U-Boot so use the mmc card instead.

```
=> efidebug boot add -b 1 TRS mmc 0:1 efi/boot/bootaa64.efi -i mmc 0:1 ledge-initramfs.
↪rootfs.cpio.gz -s 'console=ttyPS1,115200 console=tty0 root=UUID=6091b3a4-ce08-3020-
↪93a6-f755a22ef03b rootwait panic=60'
=> efidebug boot order 1
=> bootefi bootmgr
```

4.5 References

4.6 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

FIVE

CODELINE MANAGEMENT

TRS is a software development project that uses the release branch approach of Yocto. We intend to base our work and use the latest version available from the Yocto project. As mentioned before, the goal of the project is to create a reliable, stable builds by leveraging stable upstream branches and thereby reducing the risk of build and runtime issues.

5.1 Release process

By basing each release on a stable branches, the TRS project ensures reproducibility and that releases should work for a foreseeable future. The approach for making a release is as follows.

Create a manifest with stable commits: When a release is approaching, a stable snapshot is created by writing all current commits to a temporary repo xml file. This is achieved by running:

```
$ repo manifest -r -o my-release.xml
```

After this step we have well defined commits that won't change as long as we don't re-run or edit the file. This serves as the basis for our release testing.

Sanity testing: With the tagged manifests we start doing sanity testing.

Tag our own gits: After successful testing, we will tag the gits that we control and own. The tag in question is the version in the form `vMAJOR.MINOR` (from the Semantic Versioning).

Create a release branch: Next, we create a release branch for TRS in the `trs-manifest.git` repository, which will have the same name as the tag. This is important, because we use the `v<Major>.<Minor>` reference differently when checking out the release branch via repo. I.e., even though the names are the same, they have distinct uses.

Replace commits with tags: With the tags also added and the release branch created, we edit and update the `default.xml` file with the commits we got when creating the stable commits and with the tags that we've created for the gits that we own.

Update the release page: We also want to document the release. So, we need to add a section about our release at the *Changelog & Release Notes* page.

Push tags and release branch: The final step of making a release is to (git) push the tags and the branches to the upstream tree's. Once done, the release has been completed.

5.2 Release cadence

Until now releases has been a bit ad-hoc, but we plan to move to quarterly release cadence and we'll try to align them with the release of Trusted Substrate.

5.3 Branches

We provide a *Developer setup* for TRS and we also provide *Release build* for TRS. Conceptually, the only difference between these is the tagging and branching strategy. A developer build is what you get when checkout out the repo manifest without providing any branch, i.e. `-b` is not used. The developer build typically follows tip at upstream for the gits that we own and control. Other gits that we use tends to be locked to a certain tag or a commit. This is an intentional trade-off where we will have the ability to run latest on our own gits, but still not be affected by other issues that perhaps shows up in other gits.

In contrast, the release branches contain either commits or tags for all gits in their respective manifest files. Therefore, stable branches never follow the tip (which would make the non-stable).

So in summary, the `main` branch in TRS is for developers wanting to run the latest available and the branches named `v<Major>.<Minor>` are the stable branches.

CHAPTER

SIX

CONTRIBUTING

As an open source project, TRS welcomes contributions from anyone willing to submit patches that conform to the licensing rules. The primary trs.git repository uses the MIT license (see *License*). Also, for the majority of the remaining TRS projects, changes should be provided directly to the upstream source and not via TRS. TRS will adopt them when we update our manifest files to use more recent versions of the sub-projects in TRS.

6.1 Contribution Guidelines

The way to contribute is pretty much the same as what is usually done in open source projects. All patches are integrated via GitLab Merge Requests and therefore, we do not accept `*.patch` patches sent via email. That's because we want to run various regression tests on the supported devices and these tests are typically triggered by GitLab Merge Requests.

6.1.1 Forking

To be able to make a contribution, you need a GitLab account, hence start out by creating that. Once that is completed, you should fork the git where you intend to make changes. In the GitLab web interface you find a "Fork" button up to the right when you've selected a git project. Once pressed, you'll end up with your own copy of the git at your own GitLab account.

6.1.2 Merge Request

After forking the git, you'll clone the git from your own GitLab account, make your changes to the project and once you're done with the changes it's time to submit the patches. Before sending your code make sure changes have been tested locally (`make test`), squashed patches into a patch series that makes sense, written a good commit message etc. The merge request itself can be done via the GitLab web interface.

6.1.3 Commit messages

The subject line should explain what the patch does as precisely as possible. It is usually prefixed with keywords indicating which part of the code is affected, but not always. Avoid lines longer than 80 characters.

The commit description should give more details on what is changed and explain why it is done. Indication on how to enable and use a particular feature can be useful too. Try to limit line length to 72 characters, except when pasting some error message (compiler diagnostic etc.). Long lines are allowed to accommodate URLs, too (preferably use URLs in a Fixes: or Link: tag).

When it makes sense, we encourage to use other tags as well, such as:

- Tested-by: Teste R <teste@r.com>

- Acked-by: Acke R <acke@r.com>
- Suggested-by: Suggeste R <suggeste@r.com>
- Reported-by: Reporte R <reporte@r.com>

When citing a previous commit, whether it is in the text body or in a Fixes: tag, always use the format as shown in the example below, that is 12 hexadecimal digits prefix of the commit SHA1, followed by the commit subject in double quotes and parentheses.

```
crypto: RSA driver fix

This fixes e1c70d7c88ab ("crypto: drivers: se050: fix rsa encrypt/decrypt")
...
```

Review feedback

It is very likely that you will get review comments from other TRS users asking you to fix certain things etc. When fixing review comments, do:

- Add fixup patches on top of your existing branch. Do not squash and force push while fixing review comments.
- When all comments have been addressed, just write a simple messages in the comments field saying something like "All comments have been addressed". By doing so you will notify the maintainers that the fix might be ready for review again.
- When all comments have been addressed, once again rebase and squash patches into a patch series that make sense.

CHAPTER

SEVEN

LICENSE

The software is provided under the MIT license (below).

```
Copyright (c) <year> <copyright holders>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to
deal in the Software without restriction, including without limitation the
rights to use, copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice (including the next
paragraph) shall be included in all copies or substantial portions of the
Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING
FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS
IN THE SOFTWARE.
```

7.1 SPDX Identifiers

Individual files contain the following tag instead of the full license text.

```
SPDX-License-Identifier: MIT
```

This enables machine processing of license information based on the SPDX License Identifiers that are here available: http://spdx.org/licenses/

CHAPTER

EIGHT

CHANGELOG & RELEASE NOTES

8.1 v0.3 - 2023-04-19

- **Xen support**
  - It's now possible make a TRS build with Xen support. For more information about it, check the *Xen* documentation page.
- **Parsec**
  - Will now automatically be able to use OP-TEE PKCS#11 and fTPM services.
- **SDK**
  - EWAOL distro feature and OpenEmbedded SDK package has been enabled.

8.2 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.

8.3 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.

8.4 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.

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