Trusted Firmware-M Technical Overview

Feburary-2023, updated in March-2024 for URL changes.

Abstract

This document supplements the material in the Trusted Firmware-M (TF-M) User Guide (https://tf-m-user-guide.trustedfirmware.org/index.html) and covers a range of information that will help software developers to become more familiar with Trusted Firmware-M. The current version of this document reflects the technical aspects of TF-M version 1.7. The key changes from the previous version 1.6 to the current release (v1.7) are highlighted in section 7.

1 Background

If you are an embedded software developer, you might have noticed that security requirements for low cost, low power embedded systems are gaining more attention in the market. Today, many embedded systems have some form of connectivity, even though many of them are unlikely to be associated with the IoT buzzword. While taking over an IoT gadget in a home might not give a hacker much direct benefit, it could give the hacker the ability to attack (a) other devices on the same network, (b) IoT services that the device is connected to, and (c) the network infrastructure - potentially a mobile phone network if the device is connected to it. If an IoT device contains personal/valuable information, it could, of course, make the device significantly more "hacker" attractive.

In the longer term, IoT security is such a strong concern that a number of government organizations have recently investigated whether there is a need to introduce legislation to improve IoT security in IoT products. This is in stark contrast to several years ago where many embedded products had no security requirements (other than firmware read-out protection for preventing 3rd-parties from stealing the software). However, designing secure applications is not a simple task. Even when you have deployed well established security features, such as encryption/decryption of communication messages, secure boot and secure-firmware-update, a simple coding error in your application could still result in the device being compromised.

To address the challenge of creating secure embedded systems, Arm heavily invested and introduced:

- TrustZone technology in Cortex-M processors – TrustZone provides hardware enforced isolation between secure firmware and normal applications. This provides a foundation for secure software architecture.
- The Platform Security Architecture (PSA) initiative and PSA Certified program. PSA defines the technical requirements and best practices for IoT security by providing specifications and guidelines. PSA Certified provides a systematic approach which allows vendors to show that the product achieves a specific level of security.
- Trusted Firmware-M - a reference implementation of secure firmware following the API specifications from PSA Certified. The source code is available to the ecosystem as open source under the BSD-3 clause license hosted by the trustedfirmware.org community.

Using these initiatives, Arm partners have started to offer solutions which include PSA Certified devices and software that works with Trusted Firmware-M. To utilize these solutions, legacy software needs to be adapted when migrating to the new devices. This document provides an overview of Trusted Firmware-M and explains how to use it.

2 TrustZone introduction

Before we cover Trusted Firmware-M, let's start with an introduction of the TrustZone technology. Figure 1 below illustrates the partitioning between normal application code and secure firmware in a TrustZone enabled system.

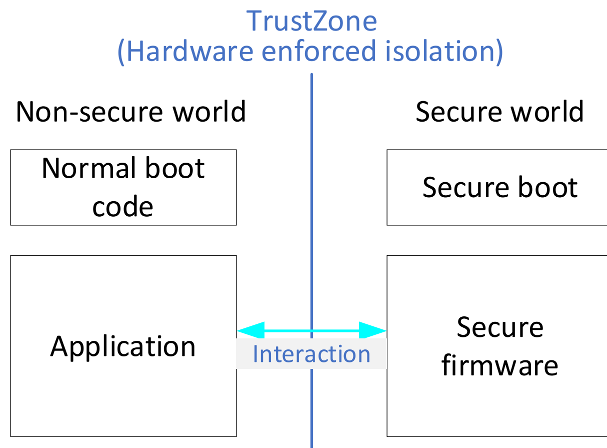

Figure 1: TrustZone provides hardware enforced isolation, while still allowing interactions between application code and secure firmware.

A common question we hear from software developers is: Why do we separate the software into Secure and Non-secure worlds when using TrustZone? And the answer is: If an application running in the Non-secure world is compromised, TrustZone prevents the attacker from accessing security critical resources, i.e., firmware update mechanisms (e.g., flash programming interface), secure storage (i.e. where the crypto keys and certificates are stored) and the random number generator (preventing the eavesdropping of random numbers used for crypto operations).

In modern IoT software there can be thousands of lines of program code, including software from third parties, potentially with software bugs lurking inside. Without a security boundary like the one provided in TrustZone, an attacker could take over the whole system once their attack payload gained privileged access permission. By having a hardware enforced isolation mechanism, an attacker should not, even when they have successfully attacked the application running in the Non-secure world, be able to fully take over a TrustZone enabled system.

You might also wonder why we need TrustZone when there are already Memory Protection Units (MPU) in legacy systems. While traditional embedded processors like the Cortex-M4 and Cortex-M7 provide MPUs for process isolation, the fact that interrupt handlers and Real-Time OS (RTOS) run in privileged state (as opposed to unprivileged state for application threads) means that if a vulnerability is present in the privileged code an attacker can bypass the MPU. As a consequence, MPUs and privilege separation are not always the bullet proof solution for IoT security. That said, an MPU is still highly beneficial in

enhancing the robustness of embedded systems and can be used in conjunction with TrustZone to give an extra level of security. Figure 2 below illustrates the use of TrustZone and MPUs together in an application.

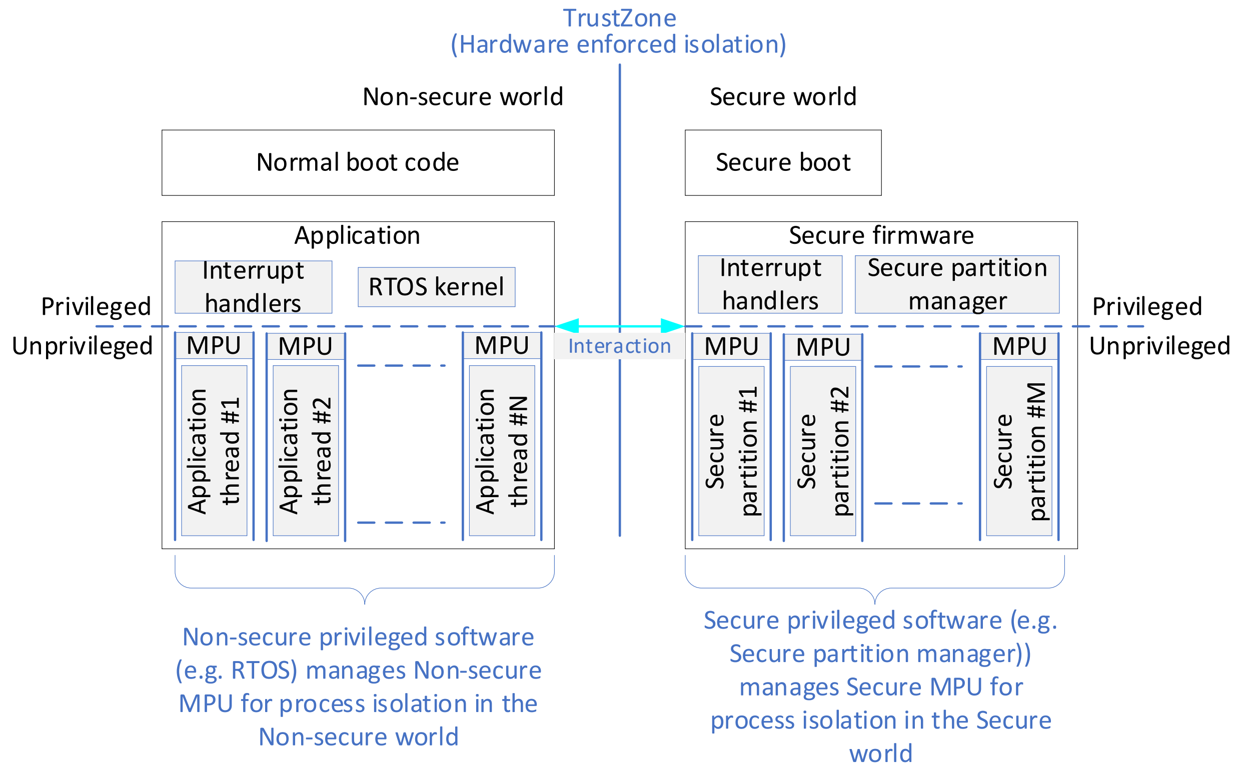

Figure 2: TrustZone and MPUs can be used simultaneously in systems that demand strong security.

The use of an MPU for isolating software components from each other in the secure firmware is optional. This configurability allows several possible software isolation levels. This topic will be covered in section 4.4.

When a Cortex-M system has TrustZone implemented, it has the following characteristics:

- The address space is partitioned into Secure and Non-secure address ranges. The partitioning mechanism consists of a programmable Security Attribution Unit (SAU) and device specific address lookup hardware connected to the Implementation Defined Attribution Unit (IDAU) interface on the processor.
- When TrustZone is implemented, there can be one MPU for the Secure world and one MPU for the Non-secure world. Both are optional and can be independently programmed and enabled.
- Secure software can access both Secure and Non-secure address ranges subject to permission outlined by the Secure MPU. Non-secure software can only access Non-secure address ranges, providing that is allowed by the Non-secure MPU. Use of each MPU is optional.
- The processor boots up in Secure state at startup and executes the secure firmware. After initialization of the security hardware and software, it then jumps into the boot code for the Non-secure world and starts the normal application.

- TrustZone for Armv8-M allows direct function calls across security states. This allows applications running in the Non-secure world to easily access Secure APIs in the Secure world.
- Security state transitions can also happen in exception entries/exits. Each interrupt can be configured as Secure or Non-secure, with some of the system exceptions also having configurable target security states. Several system exceptions are banked between security states, with the fault exception that deals with security violations (SecureFault) always targeting the Secure state.

For more information on TrustZone operations, please refer to the Armv8-M Architecture Technical Overview, which is available here:

https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/whitepaper-armv8-m-architecture-technical-overview

Of course, the security capability of a system is also dependent on the hardware design. To help silicon designers design secure systems, Arm also provides resources to guide them. For example, the following documents are available on the Arm website.

Document
Trusted Based System Architecture for Armv6-M, Armv7-M and Armv8-M (TBSA-M) https://developer.arm.com/documentation/100690/0201/
System Design with Armv8-M https://developer.arm.com/documentation/100767/0100/
Creating a System for a Secure IoT Device https://developer.arm.com/documentation/101892/0100/Build---Starting-to-develop-an-example-TBSA-M-system

3 What is Trusted Firmware-M

To help industry create secure IoT products, PSA Certified (www.psacertified.org) carried out a detailed security analysis and defined the architecture and framework needed to create secure IoT systems. A number of specifications were also defined, covering software APIs, firmware framework, etc. With these specifications, software developers can create security firmware. However, the specification on its own is not enough to enable the software ecosystem to start creating secure IoT products. Many companies do not have the time or resources to create their own secure firmware. It is also very hard for beginners in IoT software development to create secure firmware because a wide range of expertise is required. In addition to knowledge in cryptography and secure data storage, software developers also need to be familiar with additional functionalities as shown in figure 3. To overcome the aforementioned challenges, the Trusted Firmware-M project was launched.

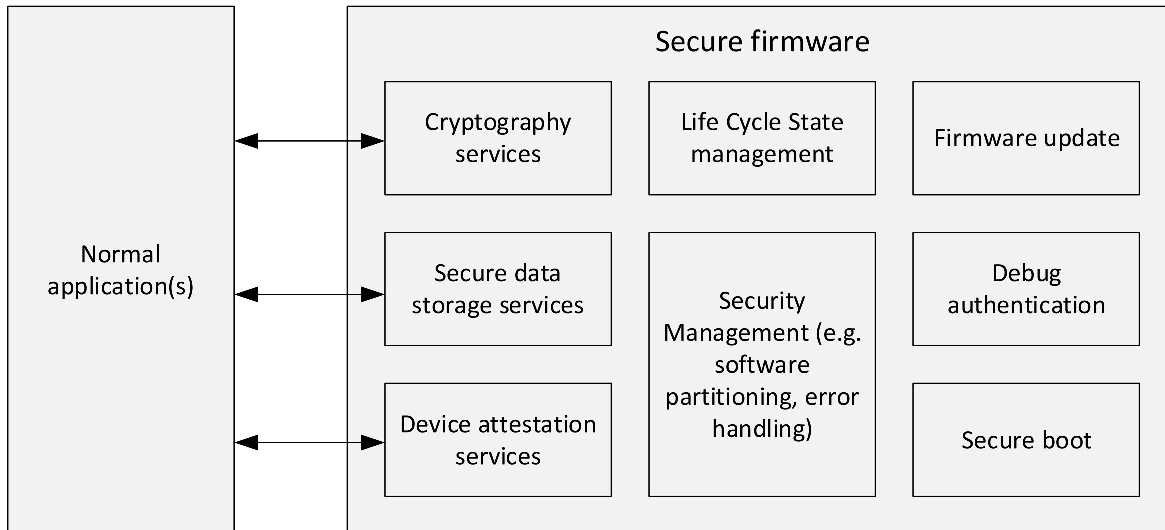

Figure 3: Example of functionalities available in typical secure firmware

Trusted Firmware-M (TF-M) is a reference secure firmware that provides functionalities as detailed in Figure 3, and it follows the specifications from PSA Certified. It provides a range of security APIs which are available for normal applications, as well as APIs for essential security management.

In a TrustZone enabled system, Trusted Firmware-M executes in the Secure world, and normal applications run in the Non-secure world (see left hand side of figure 4). Trusted Firmware-M can also work in other system configurations, e.g., a dual core environment where one of the processors is dedicated to secure firmware and where the other processor is being used for normal applications (see right hand side of figure 4). This arrangement requires additional hardware level isolation support between the two processors in the chip design to protect the secure resources from being directly accessed by normal applications.

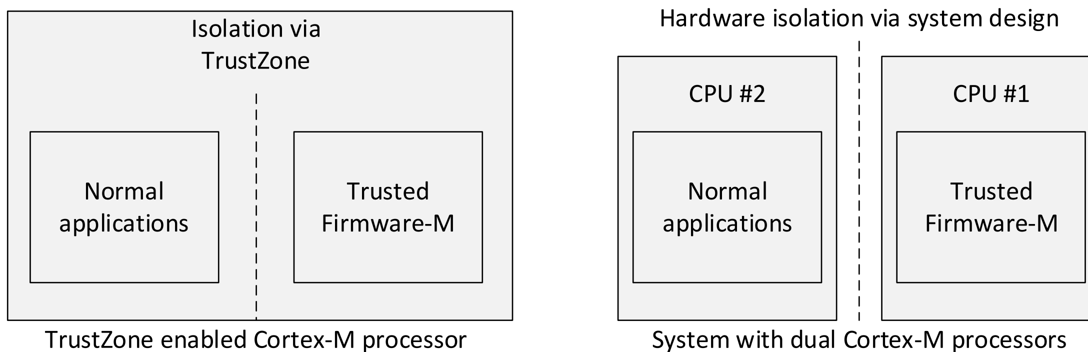

Figure 4: Two examples showing how Trusted Firmware-M can be deployed in different types of Cortex-M based devices.

Currently TF-M provides five groups of APIs (https://www.psacertified.org/development-resources/building-in-security/specifications-implementations/):

API	Website
Cryptography APIs	https://arm-software.github.io/psa-api/crypto/
Secure storage APIs	https://arm-software.github.io/psa-api/storage/ (This includes Internal Trusted Storage (ITS) and Protected Storage (PS) APIs)
Initial attestation APIs	https://arm-software.github.io/psa-api/attestation/ (This can be useful for device provisioning)
Debug authentication	https://developer.arm.com/documentation/den0101/latest
Firmware update	https://arm-software.github.io/psa-api/fwu/

These software APIs are defined by PSA Certified. Additional specifications and related materials can be downloaded from the following websites:

Material	Website
PSA Certified APIs	https://arm-software.github.io/psa-api/
PSA Certified APIs GitHub	https://github.com/arm-software/psa-api
PSA Certified Development resources	https://www.psacertified.org/development-resources/building-in-security/specifications-implementations/
Status Code API	https://arm-software.github.io/psa-api/status-code/

Trusted Firmware-M has the following key characteristics:

Trusted Firmware-M is configurable – there are many configurable parameters available. Secure software developers can define options such as:

- Which APIs to be included in the project (e.g. choice of crypto algorithms)
- The mechanism used for API call (more on this in section 4.5)
- The software isolation capability (see section 4.4)

Trusted Firmware-M is extendable - In addition to the aforementioned APIs, Trusted Firmware-M also supports custom defined secure APIs. By allowing secure firmware developers to add secure software partitions (also known as Application Root-of-Trust), these partitions can provide custom defined APIs.

TF-M is an open-source project – TF-M is a part of the Trusted Firmware project (https://www.trustedfirmware.org/) and is hosted by Linaro (https://www.linaro.org/), a not-for-profit organization dedicated in developing software for Arm-based technologies.

TF-M is in production devices – TF-M is already used in a range of devices that are in production. The project continues to add new features, new platform support and other improvements such as performance and memory size usage.

Details of TF-M can be found on the following websites:

	Web page address
TF-M home page	https://www.trustedfirmware.org/projects/tf-m/
TF-M git repository	https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/
TF-M User Guide	https://tf-m-user-guide.trustedfirmware.org/ (From this page you can also find the link to the API reference.)

TF-M roadmap	https://trustedfirmware-m.readthedocs.io/en/latest/roadmap.html

4 Trusted Firmware-M technical details

4.1 TF-M repository quick tour

While you can find the development branch of TF-M in https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/, for most projects you would likely want to use a stable release, which can be found here: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/

To compile TF-M you need a range of software tools, and these include compiler, Python and additional Python modules. The repository contains build scripts for the Arm Compiler (armclang), for the IAR Arm Compiler and for GCC. The build system is based on CMAKE. In addition, you would also need Python and a range of Python modules. The tool dependency required to compile TF-M is shown in figure 5:

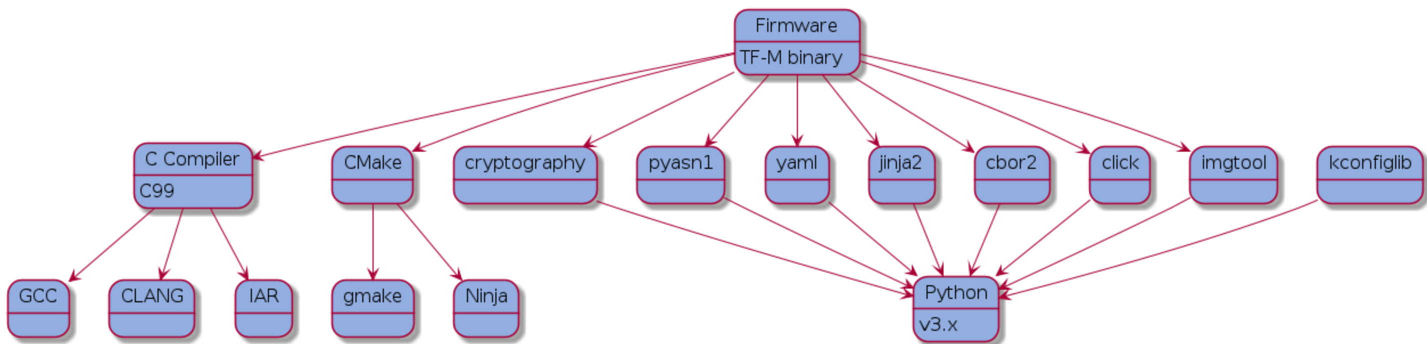

Figure 5: Tool dependency chain (Diagram from TF-M user guide)

To install the required Python modules, please use the file `tools/requirements.txt` (details are available in https://tf-m-user-guide.trustedfirmware.org/getting_started/index.html). If you are moving from TF-M 1.5 or 1.6 to 1.7, please note there are new tool dependencies. The file `tools/requirements.txt` has been updated and, as a result, you will need to install the additional tools using pip3. For example:

	Steps to install Python modules

Linux	`pip3 install --upgrade pip`
	`cd trusted-firmware-m`
	`pip3 install -r tools/requirements.txt`

Windows	`cd trusted-firmware-m`
	`pip3 install -r tools\requirements.txt`
	or
	`{python_path}\python -m pip install -r tools\requirements.txt`
	
	Note: Specifying the python path might be required if you have multiple versions of Python installed. See “Tips” below.

Tips: In some systems you might have multiple versions of Python installed and thus you will need to make sure the Python modules are installed on the specific version used by the TF-M build system. For

example, in Windows systems, you can use the “where” command to check the available Python versions in the search paths (In the example below I have two versions of Python installed):

```
cmd>where python
C:\Program Files\Python37\python.exe
C:\Users\<username>\AppData\Local\Microsoft\WindowsApps\python.exe
cmd>
```

The TF-M build system is likely to pick the most up-to-date Python release and you would therefore need to install the required python modules (listed in https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/tools/requirements.txt) using a command like:

```
cmd>C:\Users\{username}\AppData\Local\Microsoft\WindowsApps\python -m pip install -r
tools\requirements.txt
```

Similarly, on Linux systems, you can use “which” command to search for the installed Python environment(s).

Note:
The file `tools/requirements.txt` in the first release of TF-M 1.7 is missing “`windows-curses`”. You might need to install that manually using the following command:

```
> pip install windows-curses
```

Or

```
>{python_path}\python -m pip install windows-curses
```

The Getting Started page (https://tf-m-user-guide.trustedfirmware.org/getting_started/index.html) provides details on compiling and running the default test suite for AN521, an FPGA platform (or the equivalent Fixed Virtual Platform (FVP) model).

The TF-M repository contains a build system based on CMAKE. Because TF-M supports a wide range of features to assist device security for many different use cases, making all the features accessible via a simple IDE build environment is very challenging and often impractical. As a result, one of the key design choices of a TF-M project is to enable a command line build system based on CMAKE.

After the secure firmware is created, the applications running in the Non-secure world can be created using a traditional IDE. This is covered in section 5.5.

Descriptions of the TF-M source structure can be found in the following page:

https://tf-m-user-guide.trustedfirmware.org/integration_guide/source_structure/source_structure.html

Out-of-the-box, TF-M supports a number of hardware platforms. The supported hardware platforms are listed in https://tf-m-user-guide.trustedfirmware.org/platform/index.html, and the corresponding platform specific codes are located in https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/platform/ext/target. For information on the platform folder and on porting TF-M to a new hardware, please refer to:

- https://tf-m-user-guide.trustedfirmware.org/integration_guide/source_structure/source_structure.html#platform

- https://tf-m-user-guide.trustedfirmware.org/integration_guide/platform/porting_TFM_to_a_new_hardware.html

Please note, the TF-M repository pulls in external repository during the build process. For example:

- mbedcrypto (for PSA crypto implementation)
- MCU boot (for secure boot)

4.2 Configuration of TF-M

TF-M is configurable. Examples of key configuration options in TF-M include:

Item	Choice
Firmware framework model	Inter-process communication (IPC) model or Secure Function model
Isolation level	1, 2 or 3
Profile	Small, medium without Application Root-of-Trust (ARoT-less), medium, large
Cryptography support	A choice of crypto algorithms are available. See section 5.3

In addition, there are configuration parameters for:

- processor related options (e.g. support for the floating-point unit)
- regression testing,
- selecting between different types of build targets (e.g. Debug vs Release), and
- deciding whether bootloader should be built as part of the build process.

TF-M provides several mechanisms for configurating the firmware:

- Using one of the predefined profiles when building the firmware – This is one of the most common choices. See section 4.3 about TF-M profiles and section 5.2 to see an example of creating TF-M with the “small” profile.
- Using a custom defined profile when building the firmware – This could be a good solution if the configuration needed is slightly different from a pre-defined profile.
- Launching Kconfig when building the firmware – This is suitable for beginners and is currently limited to a subset of TF-M features. This is new in TF-M version 1.7. See section 5.4.
- Using the base configuration (firmware framework only) and adding features based on application requirements. You can override the default options using command line options: For example, select isolation level to 1 using -DTFM_ISOLATION_LEVEL=1.

> Note:
> There are many changes to the configuration methods in TF-M v1.7 when compared to v1.6.
> In TF-M 1.6 and older versions, most of the configuration parameters are based on CMAKE variables and CMAKE configuration files. In version 1.7, some of those configuration options are defined by C macros in C header files. This change makes it easier for the TF-M code to be compiled using traditional MCU tools. CMAKE variables are still being used for the remaining options.

4.3 TF-M Profiles

TF-M is designed to cover a wide range of application scenarios. As a result, it has many different configuration options and can be customized as small secure firmware for devices with limited memory resources, or as feature rich secure firmware. However, the wide range of configurability can make it hard for software developers to decide how best to proceed. To overcome this, TF-M has four example profiles which provide the required configurations for the most commonly used cases. These are:

- small,
- medium ARoT-less (medium without Application Root-of-Trust. This is new in TF-M v1.7),
- medium, and
- large.

These profiles help software developers configure their TF-M projects for their application requirements and for PSA certification. Use of these profiles are not mandatory. Software developers can create their own custom profiles to configure TF-M.

The following table lists the key aspects of the small, medium ARoT-less, medium and large profiles:

Profile	Small	Medium ARoT-less	Medium	Large
Firmware framework model	Secure Function (SFN) model	Secure Function (SFN) model	Inter-Process Communication (IPC) model	Inter-Process Communication (IPC) model
Isolation level	1	1	2	3
Internal Trusted Storage	ITS has decreased internal transient buffer size	Full ITS features	Full ITS features	Full ITS features
Protected storage	- (off by default)	- (off by default)	Implemented if off-chip storage device is integrated.	Implemented if off-chip storage device is integrated.
Crypto – symmetric	Supported	Supported	Supported	Supported
Crypto – asymmetric	-	Supported	Supported	Supported
Secure boot	Lightweight boot * Single image boot * Anti-rollback protection	Lightweight boot * Multiple image boot * Anti-rollback protection	Lightweight boot * Multiple image boot * Anti-rollback protection	Anti-rollback protection, multiple image boot.
Initial attestation	Based on symmetric key algorithms	Based on asymmetric key algorithms	Based on asymmetric key algorithms	Based on asymmetric key algorithms
Firmware update	- (off by default)	Supported	Supported	Supported
Software countermeasures against physical attacks	-	-	-	Supported (optional)

Information on these profiles is available as follows:

- Small : https://tf-m-user-guide.trustedfirmware.org/configuration/profiles/tfm_profile_small.html
- Medium ARoT-less (Medium without Application Root-of-Trust): https://tf-m-user-guide.trustedfirmware.org/configuration/profiles/tfm_profile_medium_arot-less.html
- Medium : https://tf-m-user-guide.trustedfirmware.org/configuration/profiles/tfm_profile_medium.html
- Large : https://tf-m-user-guide.trustedfirmware.org/configuration/profiles/tfm_profile_large.html

Each profile has a corresponding CMAKE configuration file that defines the configurations:

- Small : https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/config/profile/profile_small.cmake
- Medium ARoT-less: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/config/profile/profile_medium_arotless.cmake
- Medium : https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/config/profile/profile_medium.cmake
- Large: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/config/profile/profile_large.cmake

In TF-M, the implemented crypto algorithms are based on the selected TF-M profile, and the crypto configurations are determined according to the specific application scenario defined in the profiles. The build script selects configuration files in this location: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/lib/ext/mbedcrypto/mbedcrypto_config

Profile	Header files for cryptography support in each profile
Default (none of the small/medium/large profiles is specified)	crypto_config_default.h tfm_mbedcrypto_config_default.h
Small	crypto_config_profile_small.h tfm_mbedcrypto_config_profile_small.h
Medium / Medium ARoT-less	crypto_config_profile_medium.h tfm_mbedcrypto_config_profile_medium.h
Large	crypto_config_profile_large.h tfm_mbedcrypto_config_profile_large.h

In addition, depending on the platform, a header file tfm_mbedcrypto_config_extra_nv_seed.h could be used to define entropy related options.

When building the TF-M program image, software developers can select one of profiles detailed below based on their application requirements using a command line option:

Profile	Command line option
Small	-DTFM_PROFILE=profile_small
Medium ARoT-less	-DTFM_PROFILE=profile_medium_arotless
Medium	-DTFM_PROFILE=profile_medium
Large	-DTFM_PROFILE=profile_large

For example, to select "small" profile when building a TF-M for the Musca-S1 development board (Target "arm/musca_s1" is under platform/ext/target/ directory) on Windows, the following commands could be used:

```
>git clone https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/
>cd trusted-firmware-m
>cmake -G"Unix Makefiles" -S . -B cmake_build -DTFM_PLATFORM=arm/musca_s1 ^
   -DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake ^
   -DTFM_PROFILE=profile_small ^
   -DCMAKE_BUILD_TYPE=Debug
>cmake –build cmake_build – install
```

If there is a need to customize the features, you can, instead of using one of the existing profiles, create custom profile files in {TFM_PATH}/config/profile and use a custom profile file instead.

It is also possible to override the crypto configuration of a selected profile by, for example, including additional cryptography algorithms and editing the configuration file in `{TFM_PATH}/lib/ext/mbedcrypto/mbedcrypto_config`. An example of adding a cryptography algorithm for a "small" profile can be viewed in section 5.3.

TF-M version 1.7 introduced a base configuration. The base build includes the Secure Partition Manager (SPM) and platform support code only. This allows users to start from the TF-M skeleton/framework and enable services as required. If a profile is not specified, the settings in the base configuration would be used. Further information about the base configuration is available here:

https://tf-m-user-guide.trustedfirmware.org/configuration/index.html#base-configuration

4.4 Isolation Levels

4.4.1 Overview of isolation levels

Currently, Trusted Firmware-M provides three levels of software isolation: 1, 2 and 3, based on the Firmware Framework-M (FF-M) specification from PSA Certified. The creation of TF-M and FF-M are based on a comprehensive set of security models, and it is from these models that the required isolation boundaries are defined. TF-M conforms to the mandatory isolation rules in the FF-M.

The isolation levels describe the overall strength of isolation, with the higher level introducing more isolation boundaries. In addition to the isolation levels, software developers can, if needed, implement software isolation between PRoT (PSA Root of Trust) components. PRoT components can be placed within a single partition or contained within their dedicated partitions so that they are isolated from each other. This is an implementation defined choice and is not a requirement of the FF-M specification.

Each isolation level has its corresponding isolation rules. These rules describe the granularity for isolation, i.e. whether the constant data/code needs to be isolated or not, whether the Secure Partition Manager (SPM) should execute codes that are outside the SPM etc.

4.4.2 Level 1 isolation

This level isolates secure firmware and the secure boot from normal application(s). In a TrustZone enabled Cortex-M system, the secure firmware, secure boot and hardware resources required for security management (e.g. secure data storage, crypto engine, random number generator) are within the Secure world. Normal applications and general hardware resources are within the Non-secure world. This is illustrated in Figure 6.

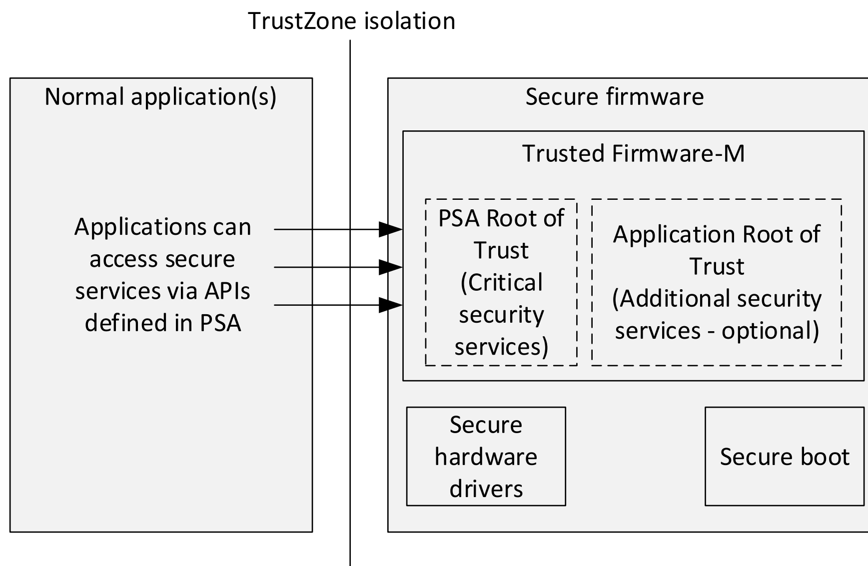

Figure 6: Level 1 isolation using TrustZone for Armv8-M

For systems with multiple processors, the isolation could be handled using system level arrangements as follows:

- Running Secure firmware and normal application(s) on different processors.
- Using system level hardware isolation to ensure that the processor running a normal application(s) is not able to access hardware resources for the Secure world.
- Normal applications using Firmware Framework-M APIs to request secure services from secure firmware. Inside the TF-M, an inter-processor communication mechanism is used to enable the processor running normal applications to communicate with the processor running the secure firmware.

For cost sensitive IoT systems, TrustZone is a better choice as it only requires a single processor.

4.4.3 Level 2 isolation

Level 2 isolation is a superset of level 1 isolation. Within the secure firmware, another isolation boundary is added to separate critical security software from non-critical security software (Figure 7 refers). The non-critical security software, also known as Application Root-of-Trust in PSA Certified terminology, could be the Protected Storage APIs, or custom defined application-level security APIs.

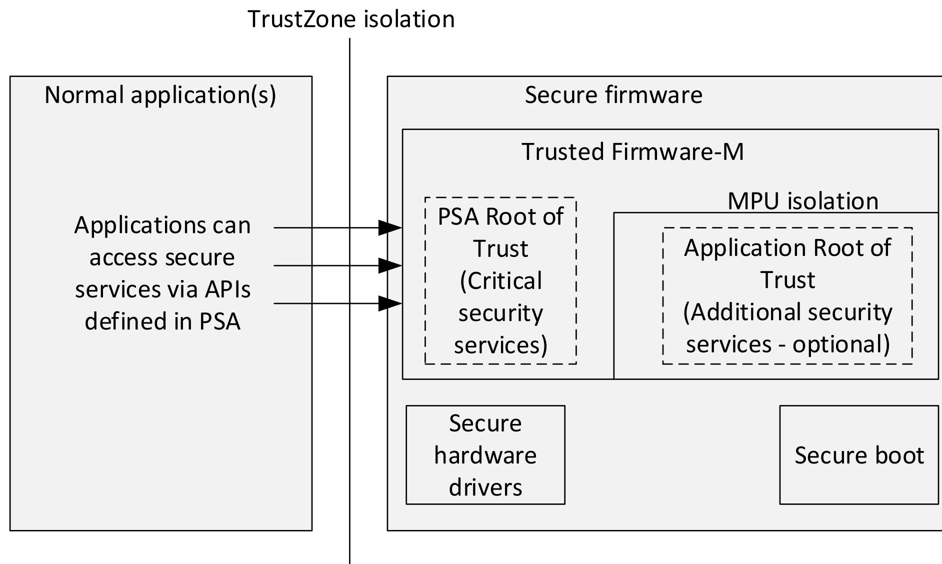

Figure 7: Level 2 isolation using MPU together with TrustZone for Armv8-M

In Cortex-M processors, the additional isolation is handled by the Memory Protection Unit (MPU). The Secure Partition Manager (SPM) within Trusted Firmware-M handles the configuration of the Secure MPU when there are context switches in the Secure side. When using this arrangement, application root-of-trust software executes in the secure unprivileged state.

4.4.4 Level 3 isolation

In some applications there can be multiple services in the Application Root-of-Trust (ARoT). Based on the requirement of level 3 isolation, these secure services need to be isolated from each other. As a result, TF-M supports isolation level 3 to provide isolation between ARoT components.

Using this isolation level, each of the Application Root-of-Trust are isolated from each other using the Secure MPU (Figure 8 refers).

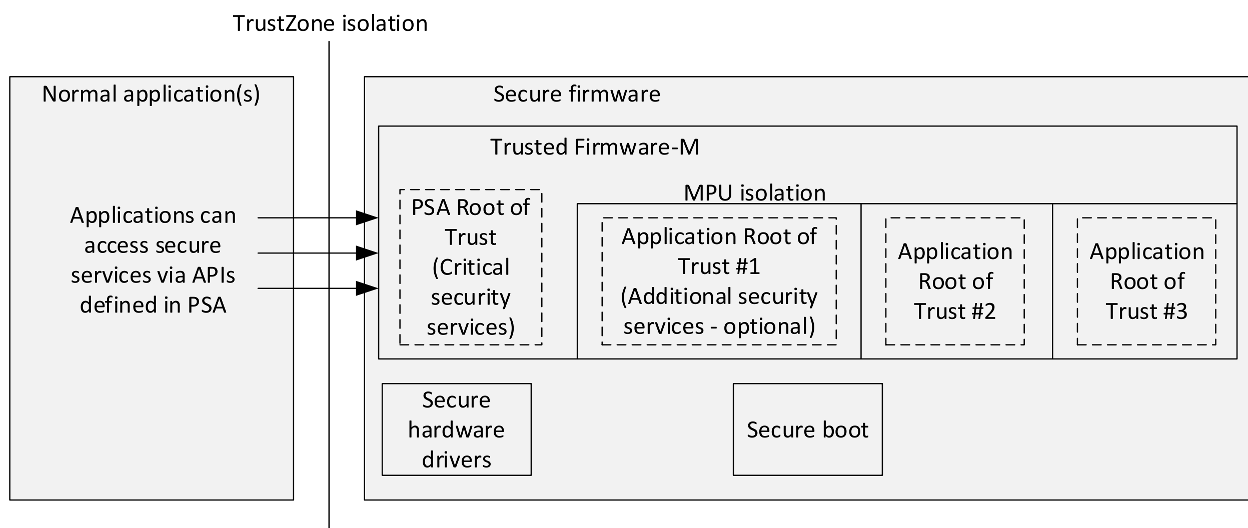

Figure 8: Level 3 isolation using MPU together with TrustZone for Armv8-M

4.4.5 Configuration of isolation level in TF-M

The isolation level in TF-M is defined using a parameter called the TFM_ISOLATION_LEVEL. Please note that this does not necessarily map to PSA Certified Levels. For devices that are certified using the PSA Certified scheme, the required isolation level depends on the PSA Certified level:

- PSA Certified Level 1 can use isolation levels 1, 2 and 3.
- PSA Certified Level 2 can use isolation level 2 and 3, or if the secure firmware does not contain any Application Root of Trust partition (i.e. ARoT-less), then isolation level 1 can be used (See note below).
- PSA Certified Level 3 must use isolation level 3.

Please note that a higher level of isolation can, due to the additional steps needed to setup the Secure MPU, increase the code size and software execution overhead.

Note: The ARoT-less protection profile was introduced in 2022-Q4. This allows devices with constrained code size capacity to support PSA Certified Level 2 when the application does not require Application Root-of-Trust. Prior to that, to reach PSA Certified Level 2, a device would have to support isolation level 2 or level 3- something that would require a considerable amount of code space.

4.4.6 Additional information on isolation support

In the current TF-M implementation, program codes of software components within PSA Root-of-Trust (PRoT) share, to simplify implementation, the same boundary. This simplification (i.e. the reduction of isolation) does not risk the program code for one software component being modified /corrupted by another. This is because program memories in most microcontrollers are based on read-only Non-volatile memories (e.g. flash memories) and cannot be changed apart from using a flash programming sequence. Consequently, PRoT's code and its constant data are visible to other secure software. However, read-write data for PRoT is isolated and protected. This arrangement enables TF-M to be used on a wider range of devices with limited hardware resources.

The secure partition manager (SPM) does not directly manage the hardware resources for enforcing the isolation. Instead, the isolation hardware resources are managed by an isolation hardware abstraction layer (HAL) APIs, with the internal details of these APIs being dependent upon the isolation level selected. When the TF-M switches between partitions, the SPM uses the isolation HAL APIs to switch the partition's isolation boundaries.

Based on the hardware used, although the implementation of isolation HAL APIs might need to be modified the SPM code can remain unchanged.

4.5 Secure API interface mechanism

4.5.1 Overview

The API specifications in PSA Certified are agnostic to the processor architecture and can be applied to a range of different devices. As a result, the API interface mechanism defined by the PSA is also hardware independent. In order to provide a generic interface that works across a wide range of devices, PSA Certified defines a software interface between Secure firmware and normal applications. This interface

is a part of the PSA Firmware Framework (PSA-FF) specification. A Cortex-M specific PSA Firmware Framework-M (PSA-FF-M) is also available.

Within the Firmware Framework specification, two interface models are provided:

- The Inter process communication (IPC) model
- The Secure Function (SFN) model – a lighter weight version that has a lower software overhead.

Different software interface models are needed due to the trade-off between security capability (isolation strength) and execution efficiency. The first version of the Firmware Framework-M (PSA-FF-M) only provides the IPC model, which although it provides great security capability has a relatively high software execution overhead. To overcome this, an API model, called Library model, was introduced into TF-M to reduce the overhead. Unfortunately, the Library model has several disadvantages. So, in FF-M version 1.1, the Secure Function (SFN) model was introduced to reduce execution overhead, resulting in the Library model being removed from TF-M version 1.7 as it was superseded by the SFN model. As a result of this development, information for the Library model is not covered in this document. Detailed analysis of the Library model can be found in appendix C of the Arm Firmware Framework-M v1.1 extensions (https://developer.arm.com/documentation/aes0039/latest)

Before I dive into the technical details of the two API models, because some understanding is required of the TrustZone mechanisms, let me start by introducing the secure API support that is available in TrustZone for Armv8-M.

4.5.2 Secure API support in TrustZone for Armv8-M

TrustZone for Armv8-M allows Non-secure software to call Secure APIs using a normal BL (Branch and Link) instruction, the same instruction that is used for standard function calls. However, to make the function call mechanism secure, the following restrictions are in place:

1) The first instruction in the Secure API being called must be an SG (Secure Gateway) instruction.
2) The address of the first instruction in the Secure API being called must be marked as Non-secure Callable (NSC).
3) As in normal secure codes, the address of the Secure API must be executable and accessible under the permission of the Secure MPU.

The SG and NSC mechanisms ensure that Non-secure codes cannot branch into the middle of a Secure function. The NSC memory attribute ensures that binary data within the Secure code that reassembles the encoding of the SG instruction cannot be mis-used as an entry point. The definition of NSC region(s) is based on the address configuration settings in the Security Attribution Unit (SAU) and in the Implementation Defined Attribution Unit (IDAU).

Instead of having lots of small NSC regions, each at the starting point of a Secure function, the Secure entry points are grouped together so that they can share a single NSC region. This is illustrated in figure 9.

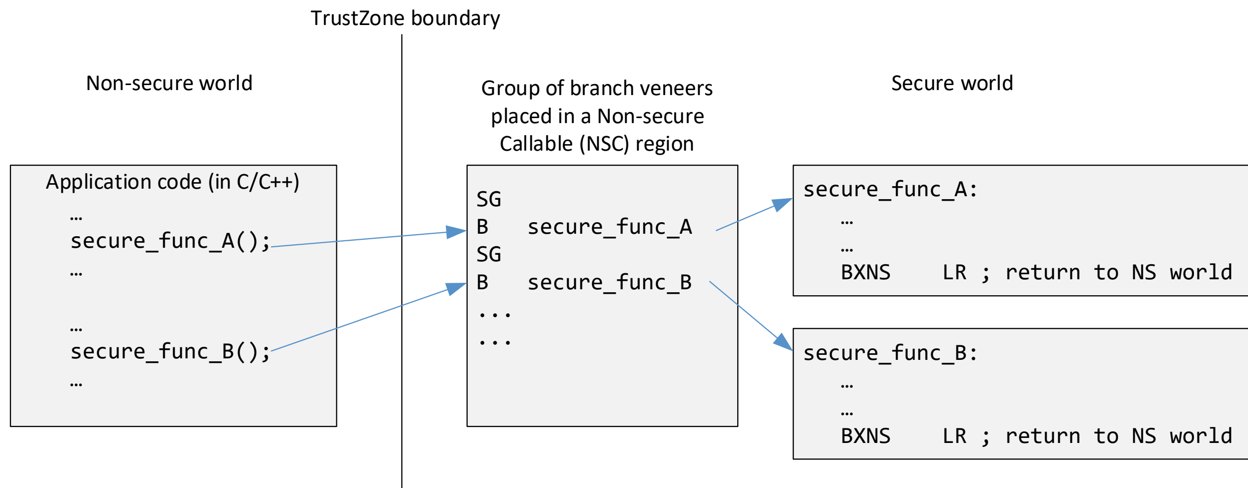

Figure 9: A branch veneer table is used to direct Secure API calls to Secure function addresses.

To help software developers create Secure APIs, Arm C Language Extension (ACLE) defines a range of tool chain features e.g. function attribute for secure APIs. The information relating to this is documented in the Cortex-M Security Extension (CMSE, https://arm-software.github.io/acle/cmse/cmse.html).

Using the CMSE support features in the C/C++ compiler, the compiler generates information about the secure APIs in the object file and ensures that the function returns use BXNS instead of a normal return instruction. The linker then generates the branch veneer table automatically using the secure APIs information provided by the compiler. This is illustrated in figure 10.

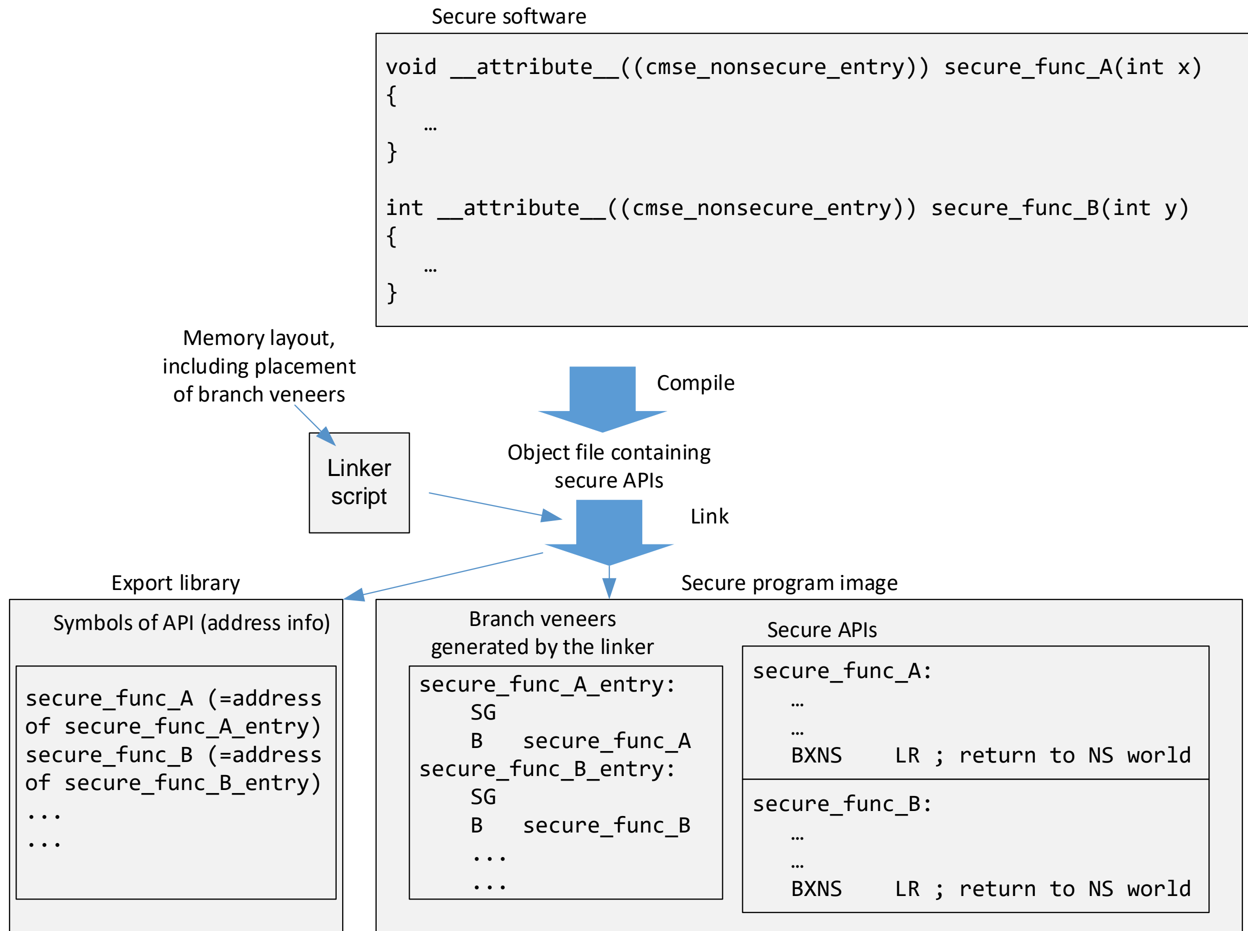

Figure 10: Branch veneer table and export library are generated by a linker during the compilation flow

In addition to the Secure program image, the linker also generates an export library. This file contains the address information of the Secure APIs and is used by a linker when developing a Non-secure program(s). The address information (i.e. symbols) enables the linker to insert right branch addresses where the Non-secure code calls Secure APIs. This is illustrated in figure 11.

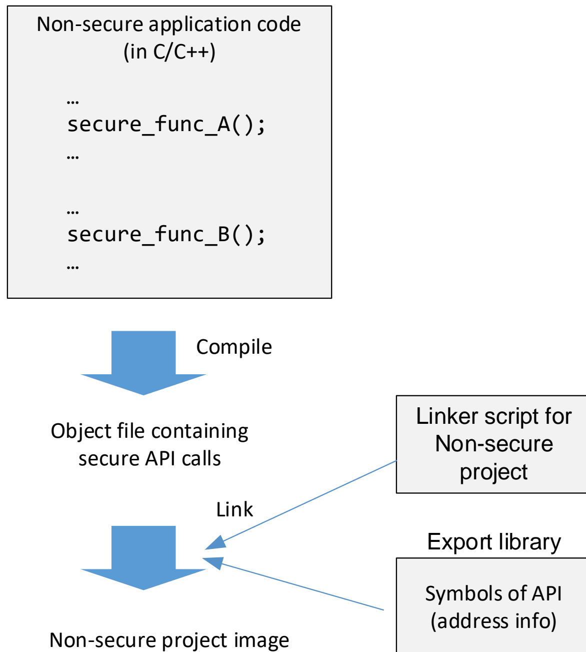

Figure 11: Export library generated from Secure firmware project is needed for building a Non-secure project

When using TF-M, after the secure project is compiled and the program image is generated, the export library (which should be shared with Non-secure software developers) is available in the following location:

trusted-firmware-m/cmake_build/install/interface/lib/s_veneers.o

Note: The secure program image can be found in trusted-firmware-m/cmake_build/install/outputs

4.5.3 TF-M Non-secure interface

Although TF-M supports a wide range of APIs (e.g. Crypto, secure storage, initial attestation, etc.), the actual interface between Secure and Non-secure program images are reduced to a few interface calls based on the PSA Firmware Framework-M (FF-M) specification. As a result, when creating a Non-secure software project that utilizes the Secure APIs, the project needs to include a few C files to handle the redirection of secure API calls. The program files required can be found in these locations:

- https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/src
- https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/include

However, instead of directly taking all the source files from these locations, Non-secure software developers should take the files from cmake_build/install/interface. This is because some of the source files might not be required. During the compilation of the secure firmware, the compilation script only copies the files that are required into the cmake_build/install/interface directory.

The use of Non-secure interface source files in secure API calls is illustrated in the following diagram:

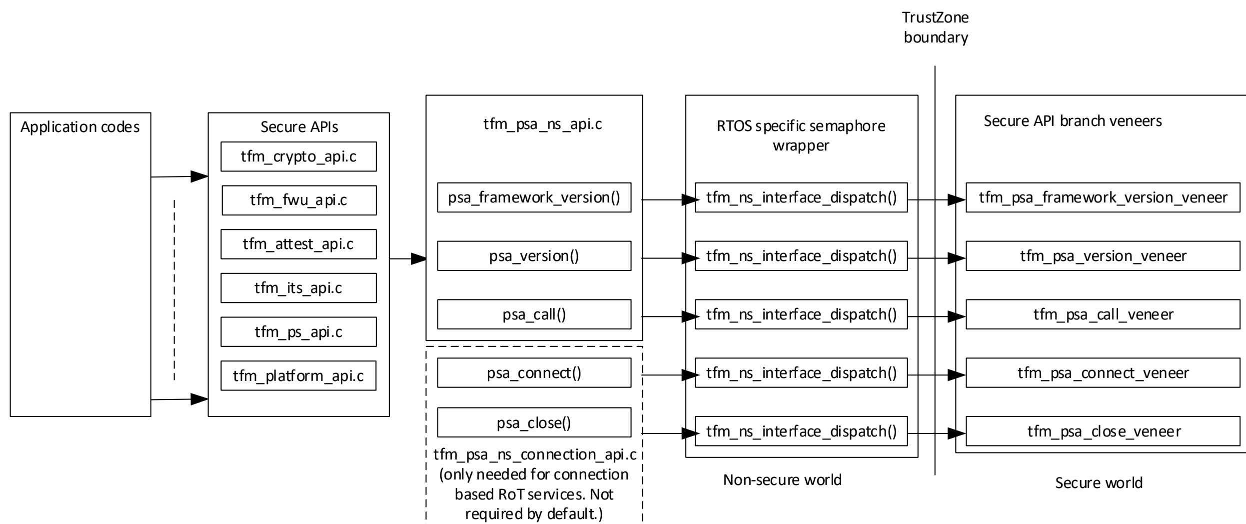

Figure 12: Secure API function call mechanism

(Note: The filenames for the Non-secure interface in TF-M v1.7 are different from TF-M v1.6).

Non-secure applications should use the APIs provided in the interface functions, which provide the interface as defined in the PSA Crypto/Secure storage/Attestation APIs.

Within the interface functions, a mutual exclusive (mutex) mechanism is added using an RTOS specific mutex wrapper. This prevents multiple Non-secure software contexts from calling secure APIs in the secure firmware at the same time.

Note:
A template of the wrapper is available in
https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/src/tfm_ns_interface.c.example
Application developers must customize the RTOS specific wrapper based on the RTOS that they are using.

A wrapper for mutex functions in FreeRTOS is available in the following location:
https://github.com/FreeRTOS/FreeRTOS-Kernel/blob/main/portable/ThirdParty/GCC/ARM_TFM/os_wrapper_freertos.c

There are several reasons not to support a multiple concurrent secure API. If a system needs to support multiple secure APIs running simultaneously, the complexity of the secure firmware and the memory size required would both increase. In real world applications, the probability of having multiple secure API calls happening at the same time is quite small. As a result, the current version of TF-M only allows one secure service at a time. If an application thread tries to call a secure API when there is already a secure service on-going, the mutex in the wrapper will delay the application thread until the on-going Secure service has been completed.

If the mutex wrapper is not used, and if a Non-secure code calls a Secure API when another secure API is still in progress (e.g. started by another Non-secure thread), the TF-M API code would return a fail status. Thus, the mutex wrapper is added to make the Secure API mechanism RTOS friendly (By delaying a new Secure API call until the on-going Secure-API call is completed).

Software developers can, if they so wish, use any other mechanism to avoid multiple secure APIs from being called simultaneously. For multiple core systems, if the secure firmware running on the "secure" processor can deal with multiple secure API requests at the same time, then it is fine to remove the mutex.

The C codes in the Non-secure interface directory also contains a file called tfm_psa_ns_connection_api.c. This file provides:

- psa_connect() – for starting an API
- psa_close() – for disconnecting

This is only needed when the secure firmware provides connection-based Root-of-Trust (RoT) services. By default, this file is not required because all the services in TF-M are stateless (Non-connection-based).

4.5.4 TF-M runtime models: IPC and SFN

In section 4.5.1 we introduced the two Secure API models available in TF-M 1.7:

- Inter Process Communication (IPC) model
- Secure Function (SFN) model (Note: This was introduced in FF-M version 1.1 to reduce execution overhead and memory footprint when compared to IPC.)

The API models define how the Secure Partition Manager (SPM) accesses/invokes services within software partitions and how the secure services are managed. Information on these two models are documented in the following page: https://tf-m-user-guide.trustedfirmware.org/integration_guide/spm_backends.html

A summary description of the two models is copied here:

- IPC backend: In this backend, the SPM and each Secure Partition have their own execution contexts, which is required to support the IPC model Secure Partitions. This also enables the SPM to provide higher isolation levels. This SPM backend acts like a multiple-process system. It can also adopt SFN model Secure Partitions.

- SFN backend: The SFN backend provides more efficient executions because it shares a single-thread execution context with all the Secure Partitions. This SPM backend acts like a single library. Therefore, it can only adopt SFN model Secure Partitions. And it does not support higher isolation levels. On the other hand, it consumes less memory compared to the IPC backend.

And in slightly more detail:

- The TF-M internal task scheduling mechanism operates differently when either of the 2 models are used

    - When using the IPC model, each partition contains a thread that "listens" to incoming service requests from the Secure Partition Manager (SPM). The SPM is responsible for dispatching API calls received from Non-secure software to the secure partitions.
    - When using the SFN model, after receiving a Secure API service call from the Non-secure software, the SPM directly calls the functions within the partitions. This arrangement avoids the delay caused by the IPC mechanisms between the SPM and the partitions.
- The IPC model enables TF-M to be used in systems that do not have a shared memory. In such systems, the information can be passed using a communication interface or a mail-box device.

Additional information about the IPC/SFN can be found here:

https://tf-m-user-guide.trustedfirmware.org/design_docs/services/secure_partition_manager.html

When creating secure firmware using TF-M, software developers can define which runtime model to use by using a CMAKE variable:

	`CONFIG_TFM_SPM_BACKEND`
IPC model	IPC
SFN model	SFN

Based on the configuration method used, this variable is setup based on

- The TF-M profile selected, or
- The configuration choice in the Kconfig menu, or
- The command line option that overrides "CONFIG_TFM_SPM_BACKEND" when building the secure firmware. For example, "-DCONFIG_TFM_SPM_BACKEND=SFN"

From a Non-secure project point of view, both the IPC and SFN models have the same interface functions as defined by the PSA Certified specifications. The API calls also route through the same RTOS specific mutex wrapper. As a result, the same Non-secure application code can work with both IPC and SFN models.

4.5.5 IPC and SFN models: Which model should I use?

To decide which API model should be used, several aspects should be considered:

- The selection of the runtime model is dependent upon the TF-M profiles:
    - Small profile or Medium ARoT-less profile: SFN model
    - Medium or large profiles: IPC model
- For devices with small memory sizes and lower processing performance, the SFN model is better as it has a lower software execution overhead. It is the default model used for a small profile. However, SFN is available only for isolation type 1. If the project requires isolation type 2 or type 3, the IPC model is required.
- The IPC model provides stronger software isolation capability and is more suitable for devices with multiple processors.

Both IPC and SFN models are compliant with the PSA Firmware Framework specification. From a Non-secure software point of view, the interface of these two models is the same, so generally speaking the choice between IPC and SFN has no impact on the design of the Non-secure software.

4.6 OS interface

4.6.1 Mutual Exclusive (Mutex)

In application environments in TrustZone enabled systems with an OS or RTOS running in the Non-secure world, there can be multiple software contexts trying to access TF-M services. Currently TF-M only supports single context, so it is necessary to restrict access to TF-M services. To prevent multiple software contexts from accessing secure API services at the same time, OS mutex operations are integrated in the Non-secure interface of the TF-M APIs.

As shown in section 4.5.3, the TF-M API calls are wrapped through a function called tfm_ns_interface_dispatch(). This function limits the application to one secure service call at a time using mutex operations and should be customized base on the RTOS being used.

From the TF-M service point of view, since there is only one service at a time, there is no need to consider multiple contexts. This reduces API complexity and memory footprint and is the preferred solution for TF-M running in microcontrollers where the memory size is limited.

Please note this mutex arrangement is for TrustZone based systems. In dual-core systems, the PSA API for the Non-secure world is implemented in another way and the arrangement can be more flexible and is platform specific. For more information on dual core implementation, please visit the following page:

https://tf-m-user-guide.trustedfirmware.org/design_docs/dual-cpu/mailbox_design_on_dual_core_system.html

An example implementation of the Non-secure interface for a dual core system can be found at:

https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/src/multi_core/tfm_multi_core_psa_ns_api.c

4.6.2 Non-secure Client Extension

In some cases, application threads running in the Non-secure world are mutually distrusted. To enable the isolation of information that Non-secure applications share with secure APIs, TF-M supports an optional Non-secure Client Extension (NSCE) for TrustZone based systems.

When using NSCE, an OS or RTOS running in the Non-secure world needs to provide TF-M with the identification of the software context using ID values. Secure services can then ensure that information associated with one application context will not be visible to another application context. For example, if Non-secure application A shared its crypto key with crypto APIs with NSID=app_a, when another Non-secure application requests access/use this key, TF-M will check if the caller has NSID==app_a, and block the request if the ID value does not match. To allow TF-M to be aware of the current context, the context switching code in the Non-secure OS needs to include TF-M context management via several dedicated APIs. (These APIs are supported in TrustZone based systems only).

By default, NSCE is disabled in TF-M, but can be enabled by setting a configuration parameter called TFM_NS_MANAGE_NSID.

An ID scheme is used to distinguish between the different software contexts running in the Non-secure world. NSCE supports two types of IDs:

- Group ID (gid, unsigned 8-bit) – this is used for separating between NSCE contexts.
- Thread ID (tid, unsigned 8-bit) – this is currently not used by NSCE for context separation. Multiple threads can share the same GID.

The GID and TID are managed by the OS running in the Non-secure world. The following APIs are defined to enable OS to communicate with NSCE:

API	Description
`uint32_t tfm_nsce_init(uint32_t ctx_requested)`	Before accessing other NSCE APIs, the Non-secure OS must call this API to request NSCE to support multiple contexts (ctx_request is the number of contexts requested). The return value determines how many contexts can be supported.
`Uint32_t tfm_nsce_acquire_ctx(uint8_t group_id, uint8_t thread_id)`	Allocates a context to a specific GID, and the API returns a token if successful, otherwise returns *TFM_NS_CLIENT_INVALID_TOKEN*.
`Uint32_t tfm_nsce_release_ctx(uint32_t token)`	If an OS thread is destroyed and no other thread is using the same context, the context can be released using this API.
`Uint32_t tfm_nsce_load_ctx(uint32_t token, int32_t nsid)`	During the OS context switch, the OS uses this API to load the context for the new thread.
`Uint32_t tfm_nsce_save_ctx(uint32_t token)`	During the OS context switch, the OS uses this API to save the context for the previous thread.

Full information of the APIs can be found at the following website address:

https://tf-m-user-guide.trustedfirmware.org/integration_guide/non-secure_client_extension_integration_guide.html

It is possible for multiple RTOS threads to share the same Group ID even though they have different Thread IDs. For example, a communication software stack/library might consist of multiple threads, with these threads sharing the same software context and crypto keys.

In theory, the interface of NSCE supports up to 256 contexts. Presently though, TF-M only supports one context at a time, but the use of NSCE is still useful for isolating secure data between contexts. Support of multiple concurrent contexts in TF-M is on the to do list of future enhancements.

When NSCE is enabled, an RTOS can utilize context management APIs as defined in CMSIS 5 (https://arm-software.github.io/CMSIS_5/Core/html/group__context__trustzone__functions.html). This requires a shim layer, which is available at this location: https://git.trustedfirmware.org/TF-M/tf-m-tests.git/tree/ns_interface/ns_client_ext/tz_shim_layer.c

Note: The current implementation of NSCE is supported in TrustZone based platforms only.

4.6.3 Additional information on OS integration

For more information about:

Integrating TF-M with RTOS – See "OS migration to Armv8-M" section in

https://tf-m-user-guide.trustedfirmware.org/integration_guide/os_migration_guide_armv8m.html

4.7 Secure boot support

Secure boot is used to ensure that a program image has not been tampered with. To use secure boot, a program image needs to be signed with a cryptographic algorithm. When the device boots up and if secure boot protection is enabled, a trusted bootloader verifies the program image and only executes the program image if it is verified.

The secure boot is separate from Trusted Firmware-M even though they both run in the Secure state in a TrustZone enabled system, or in the secure processor in a dual-core system. When the secure firmware is created, the secure boot program image and Trusted Firmware-M program image are separate from each other. The separation is needed to allow the TF-M program image to be updated without affecting the bootloader.

In Trusted Firmware-M, instead of using a new secure boot design, an open-source secure boot project called MCUboot (www.mcuboot.com, https://github.com/mcu-tools/mcuboot) was integrated. This was because MCUboot already provided the capabilities needed and was already widely used in the ecosystem. In June 2021, the MCUboot project joined the Linaro Community Projects division. If necessary, chip vendors can swap the MCUboot solution for an alternative secure boot product.

TF-M integrates 2 levels of bootloaders, both are optional:

- BL1: This is used when the device has a boot ROM. This boot loader provides the minimal features required for validation and is used to start up the second-stage bootloader BL2 in the flash memory. Because the BL1 program code is held in ROM and cannot be modified once the device is manufactured, the features in BL1 are usually kept to a minimum to reduce the risk of bugs being in the design. The following webpages provide additional information on BL1:
    - BL1 overview: https://tf-m-user-guide.trustedfirmware.org/design_docs/booting/bl1.html
    - BL1 reference code: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/bl1
- BL2: The is the main bootloader and is based on MCUboot. It provides validation of the application program image and secure firmware update. The following webpages provide additional information:
    - TF-M Secure boot: https://tf-m-user-guide.trustedfirmware.org/design_docs/booting/tfm_secure_boot.html
    - MCU boot documentation: https://github.com/mcu-tools/mcuboot/tree/main/docs
    - TF-M git repository provides a BL2 folder in https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/bl2.

During the CMAKE build process, the external MCUboot source code is pulled from the MCUboot github repository.

Not all systems require two stages of bootloaders. Many microcontroller systems that support embedded flash with flash write protection, have just a single BL2 secure boot stage.

The build process for TF-M also builds the secure boot images (this is optional). In the configuration file https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/config/config_base.cmake, the following parameters define whether the bootloader(s) should be part of the build process:

Parameter	Purpose
BL1	Specify whether BL1 boot loader is built (ON/OFF)
BL2	Specify whether BL2 boot loader is built (ON/OFF)

When running CMAKE to build TF-M, you can add command line options to override the default settings. For example:

```
cmd> cmake -G"Unix Makefiles" -S . -B cmake_build -
DTFM_PLATFORM=arm/mps2/an521 -DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake -
DCMAKE_BUILD_TYPE=Debug -DBL1=OFF -DBL2=ON
```

To sign image for secure boot, the Python imgtool is used. More examples of this are covered in section 5.5.4.

5 Creating a simple TF-M project

5.1 Building an existing TF-M regression test

The first step when trying out TF-M is to install the required toolchain. This includes the compilation toolchain (GCC / Arm Compiler / IAR Compiler), Git, Python, etc. You will also need to update the search path of the system. The full details of what is required can be found here:

https://tf-m-user-guide.trustedfirmware.org/getting_started/index.html

The aforementioned webpage also covers the second step in the process, which is to "clone" the latest Trusted Firmware-M repository using the Git tool: (an example is shown below)

```
$> mkdir test
$> cd test
$> git clone https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/
```

(Note: For production, it might be more suitable to use a stable release rather than the latest version in the repository).

In the following section, I will cover some of the basic steps. Additional information about building TF-M programs can be found in the follow webpage:

https://tf-m-user-guide.trustedfirmware.org/building/tfm_build_instruction.html

Assuming we are going to use gcc, and the target is the Arm AN552 FPGA image (an FPGA image for Arm MPS3 FPGA board with a Cortex-M55 processor inside), we run the compilation using the following commands (I have assumed that Windows 10 is being used):

```
$> cd trusted-firmware-m
$> cmake -G"Unix Makefiles" -S . -B cmake_build -DTFM_PLATFORM=arm/mps3/an552
-DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake -DCMAKE_BUILD_TYPE=Debug
-DTEST_S=ON -DTEST_NS=ON
$> cmake --build cmake_build -- install
```

If a different toolchain is required, this can be achieved by changing the TFM_TOOLCHAIN_FILE parameter as follows:

Toolchain	Parameter/Variable
GCC	-DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake
Arm Compiler 6 (armclang)	-DTFM_TOOLCHAIN_FILE=toolchain_ARMCLANG.cmake
IAR Compiler	-DTFM_TOOLCHAIN_FILE=toolchain_IARARM.cmake

The whole build process can take several minutes. After the compilation is finished, the output signed program image (binaries) are copied to the trusted-firmware-m\cmake_build\install\outputs. Once done, follow the instructions in AN552 page (link below) to copy the signed images to the MPS3 storage and test the generated binary files:

https://tf-m-user-guide.trustedfirmware.org/platform/arm/mps3/an552/README.html

The execution output should show that the system boots up in the bootloader, verifies the program images, initializes Trusted Firmware-M, switches to the Non-secure world and starts the regression tests.

By default, the compilation flow enables a range of features and the regression tests in the compiled program tests those features. If you are using a microcontroller device with limited memory, the regression tests might not fit. If this happens, you should use the small profile instead to reduce the memory usage (this will be shown in the next step).

The aforementioned example command used the following parameters to enable the regression tests:

-DTEST_S=ON -DTEST_NS=ON

The parameter:

- TEST_S enables all the secure regression suites supported on the platform.
- TEST_NS enables all the non-secure secure regression suites supported on the platform.

These parameters are used for testing only and should be removed for production.

TF-M also allows users to select/de-select regression test suites individually. For example, crypto, ITS, a Non-secure test suite or a Secure test suite. This arrangement is very useful if users just want to test a specific feature, a secure partition or when the platform is resource constrained.

TF-M supports out-of-tree regression test builds (i.e. use of regression tests that are not part of the git repository). It enables users to integrate out-of-tree extra test suites with TF-M to easily and flexibly

perform platform specific/application specific test suites. More information on this subject can be found in the “Out-of-tree regression test suites” section in the following document:

https://git.trustedfirmware.org/TF-M/tf-m-tests.git/tree/docs/tfm_test_suites_addition.rst

If you need to view the generated messages for each step in the compilations, add --verbose to the build command (i.e. in the second step). For example:

```
$> cmake --build cmake_build --verbose  -- install
```

5.2 Creating a TF-M firmware image with the TF-M Profile Small

In this step, I will show you how to create a TF-M firmware image:

- With a TF-M Profile Small (using SFN model), and
- Which does not include the regression tests.

To ensure we do not end up using the old configuration files, we should first remove the old cmake_build directory (which was automatically generated) and then run the following commands:

```
$> cd trusted-firmware-m

$> rmdir cmake_build /s    (Remove cmake_build for Windows)
or
$> rm -r cmake_build    (Remove cmake_build for Linux)

$> cmake -G"Unix Makefiles" -S . -B cmake_build -DTFM_PLATFORM=arm/mps3/an552
   -DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake -DCMAKE_BUILD_TYPE=Debug
   -DTFM_PROFILE=profile_small

$> cmake --build cmake_build -- install
```

After the compilation is finished, the program image should, as before, be downloaded to the development board and executed. However, this time the program execution seems to have stopped after entering the Non-secure world, as shown in the execution message below. This is expected, as we have removed the regression tests from the build.

```
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT SECURE
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT SECURE
```

```
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.7.0+128ada80d
Creating an empty ITS flash layout.
[INF][Crypto] Provisioning entropy seed... complete.
[DBG][Crypto] Initialising mbed TLS 3.2.1 as PSA Crypto backend library... complete.
Non-Secure system starting...
```

Because the regression tests were disabled, the Non-secure application has nothing to do – apart from starting up an RTOS and then entering an “IDLE” state. To ensure that there is a working TF-M image, we should modify the Non-secure application so that the code calls a Secure function, i.e. “psa_framework_version()”.

To carry out this modification, we need to locate the Non-secure application code. The “main()” function of the Non-secure application is part of the TF-M test suite, which is in a separate git repository as follows:

https://git.trustedfirmware.org/TF-M/tf-m-tests.git

During the build process, the external git repositories (including the TF-M test suite, MCUboot and mbedTLS) are pulled into the cmake_build directory. You can find them at the following location:

trusted-firmware-m/cmake_build/lib/ext

The test application can be found in the trusted-firmware-m/cmake_build/lib/ext/tfm_test_repo-src/app, as follows:

- main_ns.c – starts an OS and an application thread “test_app()” (inside test_app.c).
- test_app.c – contains test_app().

We can modify test_app.c to

- Add #include “stdio.h” for printf function
- Add calls to psa_framework_version()

The highlighted text below is the added code for calling the psa_framework_version() function.

```
/*
 * Copyright (c) 2017-2022, Arm Limited. All rights reserved.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 *
 */

#include "test_app.h"
#include "tfm_log.h"
#include "stdio.h"

extern uint32_t psa_framework_version(void);
```

```
...
/**
 * \brief Services test thread
 *
 */
__attribute__((noreturn))
void test_app(void *argument)
{
   uint32_t  v_temp;
   UNUSED_VARIABLE(argument);

   v_temp = psa_framework_version();
   printf ("PSA Framework version = %d\r\n", v_temp);
   ...
```

After the code is edited, we recompile the code by running

```
$> cmake --build cmake_build -- install
```

After uploading the test program image to the platform and restarting it, the following output message should be displayed after the Non-secure software is executed:

```
[INF] Starting bootloader
[INF] Beginning BL2 provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT SECURE
[INF] Swap type: none
[INF] Bootloader chainload address offset: 0x0
[INF] Jumping to the first image slot
[INF] Beginning TF-M provisioning
[WRN] TFM_DUMMY_PROVISIONING is not suitable for production! This device is NOT SECURE
[Sec Thread] Secure image initializing!
TF-M isolation level is: 0x00000001
Booting TF-M v1.7.0+128ada80d
Creating an empty ITS flash layout.
[INF][Crypto] Provisioning entropy seed... complete.
[DBG][Crypto] Initialising mbed TLS 3.2.1 as PSA Crypto backend library... complete.
Non-Secure system starting...
PSA Framework version = 257
```

For reference, the function prototype of these APIs is as follows:

```
uint32_t psa_framework_version(void);

uint32_t psa_version(uint32_t sid);
```

Note: The function “psa_version()” is not used in the example because it might not be available for a minimal configuration. Definition of SID (Service ID) can be found in table 37 of this page: https://tf-m-user-guide.trustedfirmware.org/integration_guide/services/tfm_secure_partition_addition.html

Tip: In TrustZone based systems with an OS running in the Non-secure state, the PSA APIs must be called in OS threads only. So, instead of adding the codes to main() in main_ns.c, it is added to test_app(). Each PSA API goes through a wrapper function called tfm_ns_interface_dispatch(), with this wrapper function requesting a mutex before calling the entry point. Because the mutex function only works inside OS threads, the code could fail or get stuck if it is called inside the main() function.

5.3 Customizing crypto options

In many instances there is a need to customize the cryptography functions in the secure firmware. For example, the TF-M Profile Small does not include asymmetric cryptography support. If an asymmetric crypto operation is needed, you can either use another TF-M profile (e.g. Medium ARoT-less), or customize the cryptography options.

In the configuration files for a TF-M Profile Small in the trusted-firmware-m\config\profile directory, the following default options are defined:

Option	Default state	Note
CRYPTO_ASYM_SIGN_MODULE_ENABLED (This is defined in config_profile_small.h)	0	When this is 0 (default), the PSA Crypto asymmetric key signature module is disabled. To enable the asymmetric key signature module, this value must be set to 1.
CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED (This is defined in config_profile_small.h)	0	When this is 0 (default), the PSA Crypto asymmetric key encryption module is disabled. To enable the asymmetric key encryption module, this value must be set to 1.

In addition, there is an Attestation configuration which only configures the attestation service and which has no impact on the crypto service:

Option	Default state	Note
SYMMETRIC_INITIAL_ATTESTATION (This is defined in profile_small.cmake)	ON	By default, symmetric crypto is used for initial attestation. To use asymmetric crypto for initial attestation, this parameter should be set to OFF.

You can override the default options by creating a custom defined profile file and editing the values.

For more information about configurations and the order of configuration options, please refer to the following page:

https://tf-m-user-guide.trustedfirmware.org/configuration/build_configuration.html

In addition, you might need to edit the corresponding mbedcrypto configuration file to enable the required algorithm(s). For the TF-M Profile Small, the configuration file is in this location:

trusted-firmware-m\lib\ext\mbedcrypto\mbedcrypto_config\crypto_config_profile_small.h

The configuration options in this header file are specific to the mbedcrypto software component. Although the mbedcrypto library is part of the secure firmware, it is independent from the TF-M and therefore the configuration file is separate.

When CRYPTO_ASYM_SIGN_MODULE_ENABLED is set to 1, the mbedcrypto configuration must support at least one asymmetric crypto algorithm. One method is to use the same configuration as in the Profile Medium. To match the same configurations used in the Profile Medium, the following crypto algorithm macros in the mbedcrypto configuration file need to be set to 1:

- PSA_WANT_ALG_ECDSA
- PSA_WANT_ALG_DETERMINISTIC_ECDSA
- PSA_WANT_ALG_ECDH
- PSA_WANT_ALG_HKDF
- PSA_WANT_KEY_TYPE_ECC_KEY_PAIR
- PSA_WANT_KEY_TYPE_ECC_PUBLIC_KEY
- In addition, at least one of the asymmetric crypto options needs to be set (e.g. PSA_WANT_ECC_SECP_R1_256)

When CRYPTO_ASYM_ENCRYPT_MODULE_ENABLED is set to 1, the following crypto algorithm macros need to be set to 1:

- Either PSA_WANT_ALG_RSA_OAEP or PSA_WANT_ALG_RSA_PKCS1V15_CRYPT
- PSA_WANT_KEY_TYPE_RSA_KEY_PAIR
- PSA_WANT_KEY_TYPE_RSA_PUBLIC_KEY

The SYMMETRIC_INITIAL_ATTESTATION parameter only affects the attestation. The choice depends on the provisioning requirements. For devices with a small memory size, initial attestation based on symmetric key algorithm is preferred. If asymmetric key algorithm is needed, using the medium ARoT-less profile would be more straight forward.

Potentially, you might also need to adjust other configuration parameters related to crypto support. For example:

- CRYPTO_ENGINE_BUF_SIZE
- CRYPTO_CONC_OPER_NUM
- CRYPTO_IOVEC_BUFFER_SIZE

Details of these parameters can be found in this page:

https://tf-m-user-guide.trustedfirmware.org/design_docs/services/tfm_crypto_design.html

For a device with limited memory, and when the application does not need to carry out many cryptography operations, the maximum number of concurrent crypto operations can be reduced; e.g. by setting CRYPTO_CONC_OPER_NUM to 1.

5.4 Configuring TF-M using Kconfig

5.4.1 Overview

Version 1.7 of TF-M introduced Kconfig support to make it easier for beginners to configure TF-M. This arrangement allows users to browse the available options through a user interface and to easily adjust the settings. There are two ways to use Kconfig in TF-M.

5.4.2 Method 1: Launch Kconfig when running CMAKE.

This method combines the Kconfig configuration step with the CMAKE build process. When using this method, a command line option "-DUSE_KCONFIG_TOOL" is passed to the CMAKE script resulting in the script launching the Kconfig graphical user interface (GUI) during the CMAKE execution process (Figure 13 refers). After you configure the TF-M options and save the settings, quit Kconfig and leave the CMAKE script to continue generating the build environment.

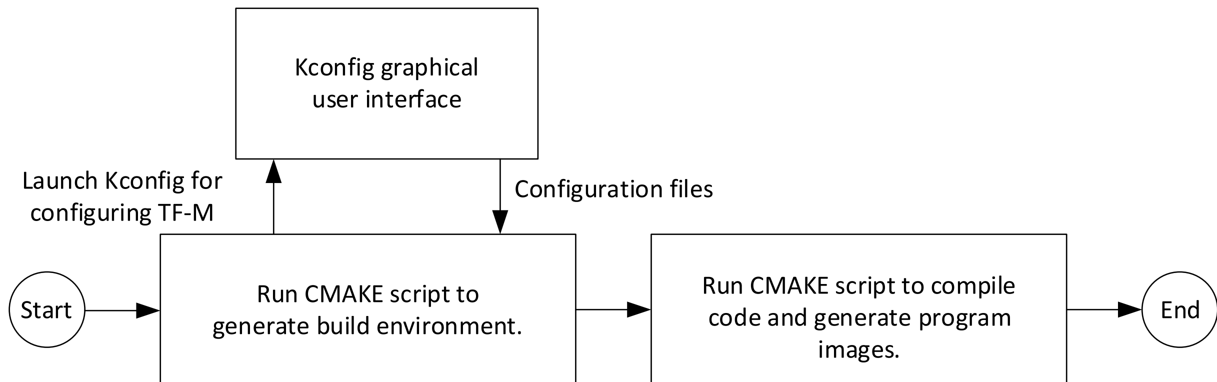

Figure 13: Running Kconfig as part of the CMAKE build process.

An example command line for running CMAKE script and configuring TF-M using Kconfig GUI is as follows:

```
>cd trusted-firmware-m
>cmake -G"Unix Makefiles" -S . -B cmake_build -DTFM_PLATFORM=arm/mps3/an5552 ^
  -DUSE_KCONFIG_TOOL=1 -DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake  ^
  -DCMAKE_BUILD_TYPE=Debug
```

Shortly after the script starts, Kconfig GUI is launched as shown in the following example screen shot:

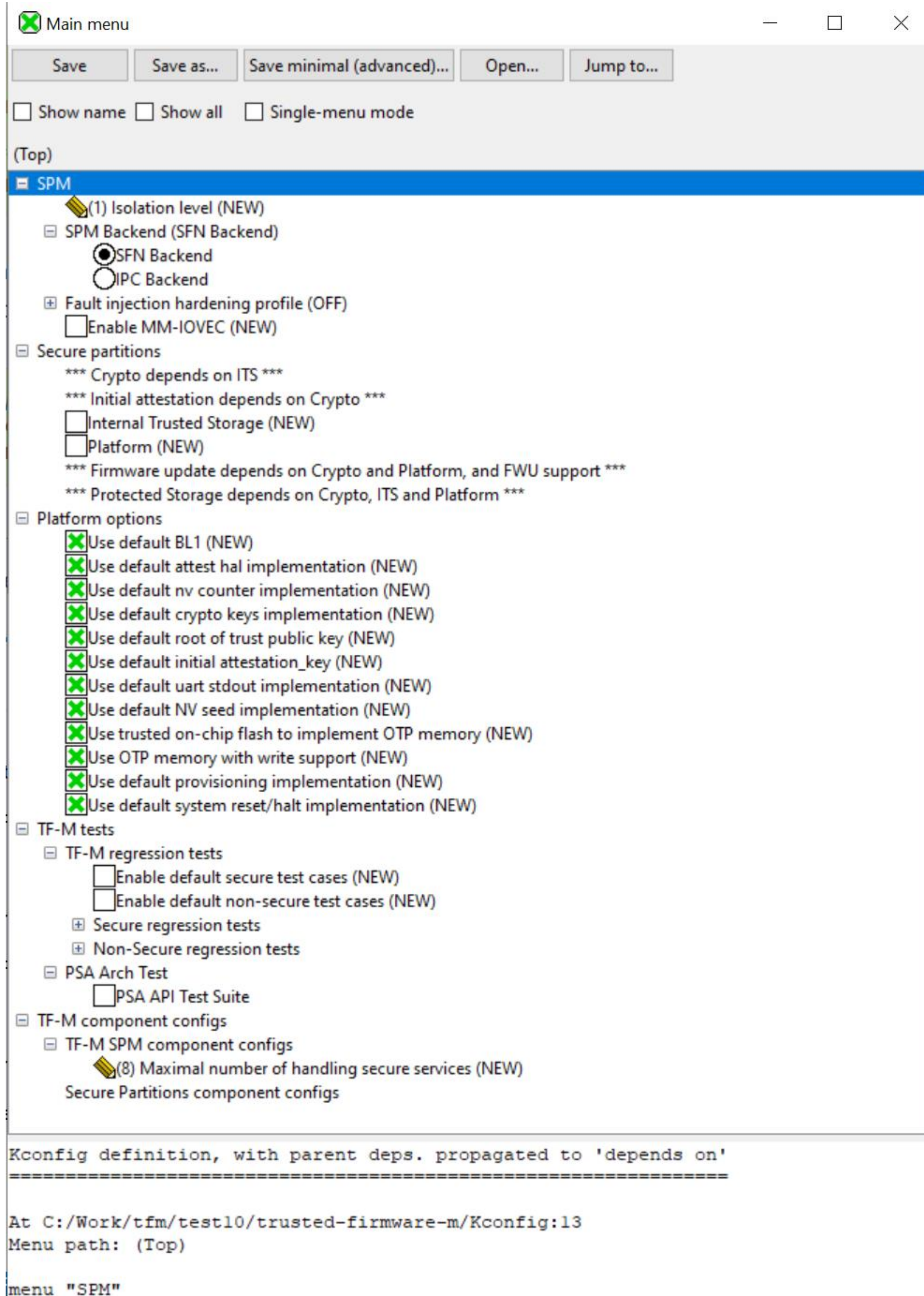

Figure 14: Kconfig graphical user interface.

After adjusting the configuration options, click on “Save” and then quit Kconfig: the build script will continue, allowing you to compile the secure firmware (e.g. cmake --build cmake_build -- install).

5.4.3 Method 2: Launch Kconfig as a separate step before running the CMAKE script.

It is possible to launch Kconfig as a separate step (step 1) and then use CMAKE to generate the build environment (step 2) and compile the code (step 3). This is shown in Figure 15. This method is more suitable for test environments where a set of configuration files need to be reused multiple times (e.g. for regression testing).

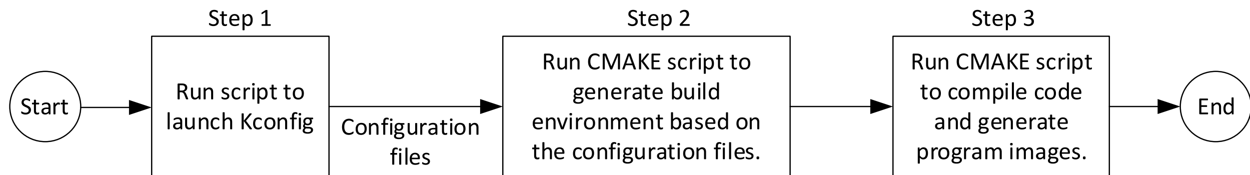

Figure 15: Running Kconfig as a standalone step before running CMAKE.

The process for method 2 is as follows:

Step 1: Launch Kconfig using the script “tools/kconfig/tfm_kconfig.py”.

```
>cd trusted-firmware-m
# Execute one of the following commands:
# (1) For text-based user interface
>python3 tools/kconfig/tfm_kconfig.py -k Kconfig -o my_config -u tui
# (2) For graphic user interface
>python3 tools/kconfig/tfm_kconfig.py -k Kconfig -o my_config -u gui
```

After saving the configuration and exiting the Kconfig user interface, the configuration files are created in a directory (I used “my_config” for the directory name in the above example. You can change that to another directory name if you prefer).

Step 2: Run CMAKE using the generated configuration as input.

The following shows an example command to run CMAKE with the configuration files as input:

```
> cmake -G"Unix Makefiles" -S . -B cmake_build -DTFM_PLATFORM=arm/mps3/an552   ^
  -DTFM_EXTRA_CONFIG_PATH=my_config/project_config.cmake ^
  -DPROJECT_CONFIG_HEADER_FILE=my_config/project_config.h ^
   -DTFM_TOOLCHAIN_FILE=toolchain_GNUARM.cmake -DCMAKE_BUILD_TYPE=Debug
```

Step 3: Compile program code

After generating the build environment, start the compilation (e.g. cmake --build cmake_build – install).

5.4.4 Limitations

At this stage of TF-M version 1.7, there are some restrictions with the Kconfig support. These are:

- Only isolation level 1 is supported.
- The configuration generated does not align with options from the command line, therefore options like test configuration (e.g. -DTEST_S=ON -DTEST_NS=ON) should not be used.

Additional information relating to Kconfig support is available from the following page:

https://tf-m-user-guide.trustedfirmware.org/configuration/kconfig_system.html

5.5 Creating a Non-secure application with a traditional IDE

5.5.1 Overview

After creating a working secure firmware image, many software developers will then work on their application software projects that run in the Non-secure world. For such software development, it is usual to use a traditional or cloud-based IDE because those IDE's offer better debug capabilities.

To create a Non-secure application, software developers need to include the following items in their Non-secure software project:

- The export library (e.g. s_veneers.o)- which is generated when the Secure firmware is compiled.
- The Non-secure interface codes for TF-M
  Note 1: Instead of taking the files directly from https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/, the file generated during the building of the secure program image should be used. This is available in the cmake_build/install/interface directory.
  Note 2: A CMSIS-PACK for the TF-M Non-secure interface codes is currently being prepared and will make it easier to integrate the TF-M interface.

In addition, software developers also need to make sure that:

- The memory layout for their Non-secure project does not conflict with the Secure memory.
- The generated software program image is signed after the Non-secure project is compiled. If it is not, the secure boot loader will not load the program.

To make the software development easier, many software stacks for IoT applications are available as CMSIS-PACK: This means they can easily be integrated into software projects when using development environments like Keil Microcontroller Development Kit (MDK) and Keil Studio.

5.5.2 Integration of the Non-secure interface

Typically, software developers when building their project need to include the following C source files in cmake_build/install/interface/src. For example, you might see the following files after running the TF-M build scripts (Note: some of these files might not exist if certain features are not enabled):

```
cmake_build/install/interface/src/tfm_attest_api.c
cmake_build/install/interface/src/tfm_crypto_api.c
cmake_build/install/interface/src/tfm_its_api.c
cmake_build/install/interface/src/tfm_ps_api.c
cmake_build/install/interface/src/tfm_psa_ns_api.c
cmake_build/install/interface/src/tfm_fwu_api.c
cmake_build/install/interface/src/tfm_platform_api.c
```

In addition to the above, Software developers should also include:

- The Secure API veneer (cmake_build/install/interface/libs_veneers.o)

- The C header files in cmake_build/install/interface/include and the subdirectories to the search paths for “include” files.
- A modified version of an OS specific wrapper (See https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/src/tfm_ns_interface.c.example).

To assist integration with FreeRTOS, a mutex function wrapper is available at this GitHub location:

https://github.com/FreeRTOS/FreeRTOS-Kernel/tree/main/portable/ThirdParty/GCC/ARM_TFM

If needed, a Non-secure software developer is able to examine the available entry points in the export library (s_veneers.o) by listing the available symbols. For example, if using Arm Compiler 6:

$>fromelf -s s_veneers.o

If using GCC:

$> arm-none-eabi-objdump -x s_veneers.o

The command lists the available symbols in the export library. This can be useful when the software developer is considering using connection-based services which require psa_connect() and psa_close() functions.

5.5.3 Memory layout

Software developers creating Non-secure software must ensure that the memory layout settings does not conflict with the memory usage for the Secure firmware. If you are using an existing port of TF-M, you can identity the memory layout from the configuration header files in the target folders. For example:

- For AN552 (one of the FPGA images on the Arm MPS3 FPGA), the header files describing the memory layout are in the trusted-firmware-m/platform/ext/target/arm/mps3/an552/partition
- For STM32l562, the header files describing the memory layout are in the trusted-firmware-m/platform/ext/target/stm/stm32l562e_dk/include

Software developers could, if they wish, use the memory layouts of the example Non-secure projects I have detailed above as a reference and use the same arrangement for their own application projects.

5.5.4 Signing of the generated image

After the program image (binary file) is created, it needs to be signed before being used. In the CMAKE script in the TF-M project, two Python script files are used. Figure 16 shows the flow of image signing using those scripts and the corresponding input/output parameters.

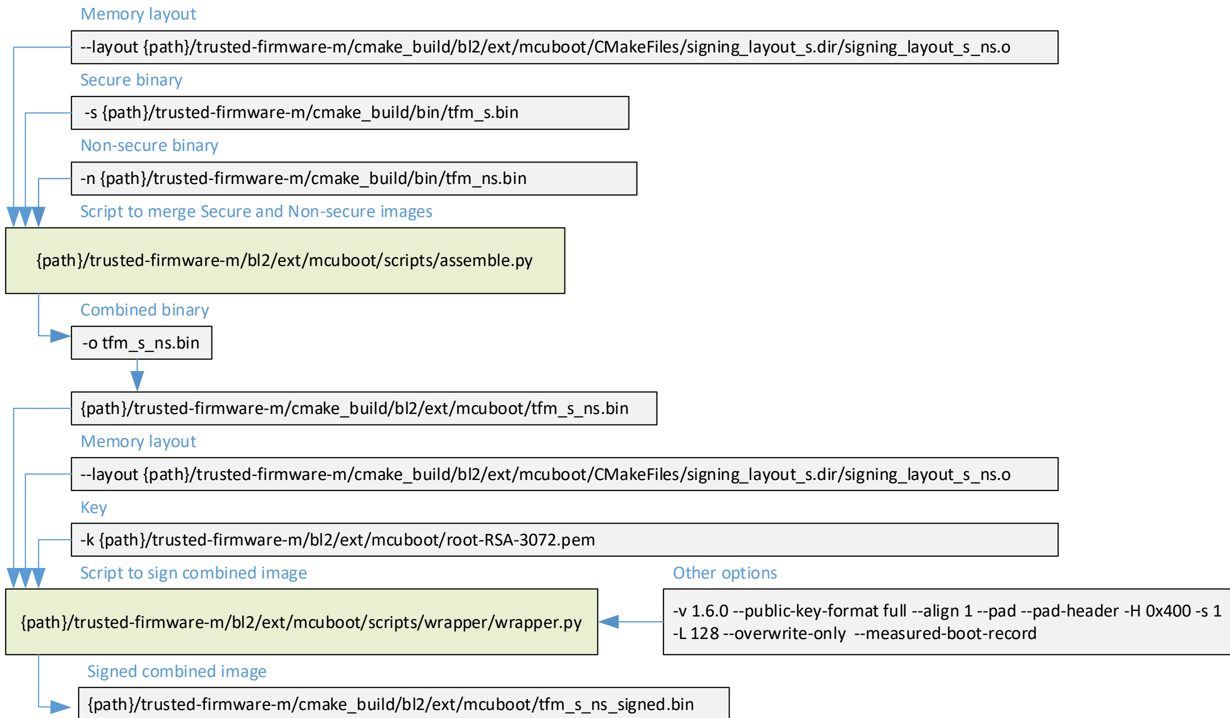

Figure 16: TF-M image signing flow.

The first Python script (assemble.py) is used to merge two binary images into one. The reason for this is platform specific: The flash programming algorithms in some devices might require both Secure and Non-secure images to be programmed together.

The second Python script (wrapper.py) is a wrapper for imgtool, a Python module for image signing. This script takes the same memory layout file and passes the details to imgtool.

Once the software developer has created a new Non-secure program image (in binary format), the signed image can be regenerated using the aforementioned scripts:

```
(Assumed using Windows)
python3.10.exe bl2\ext\mcuboot\scripts\assemble.py ^
  --layout
..\..\..\..\cmake_build\bl2\ext\mcuboot\CMakeFiles\signing_layout_s.dir\signing_layout_s_ns.o ^
  -s cmake_build\bin\tfm_s.bin ^
  -n cmake_build\bin\tfm_ns_new.bin ^
  -o tfm_s_ns_new.bin

python3.10.exe bl2\ext\mcuboot\scripts\wrapper\wrapper.py -v 1.6.0 ^
  --layout
..\..\..\..\cmake_build\bl2\ext\mcuboot\CMakeFiles\signing_layout_s.dir\signing_layout_s_ns.o ^
  -k bl2\ext\mcuboot\root-RSA-3072.pem ^
  --public-key-format full --align 1 --pad --pad-header -H 0x400 -s 1 -L 128 --overwrite-only ^
  --measured-boot-record ^
  tfm_s_ns_new.bin  ^
  tfm_s_ns_signed_new.bin
```

It is possible to just sign the Non-secure image. Usually this arrangement is used for microcontrollers where the Secure and Non-secure flash can be independently programmed. An example of using this arrangement is covered in the next section (section 5.6).

5.6 Example based on AWS MQTT service and TF-M

To make the best use of TF-M, the application codes need to utilize the security APIs provided by TF-M. An example of this is that the AWS IoT software library now supports PSA APIs and therefore can take advantage of TF-M. A demonstration of this is posted on Github: https://github.com/MDK-Packs/TrustZone. This repository provides multiple demos including Authentication (Provisioning) and MQTT data transfer.

> Note: Because currently this demo is using an older version of TF-M the Non-secure interface files are different from the latest ones in the TF-M repository.

The AWS IoT library uses a C macro called MBEDTLS_TRANSPORT_PSA to enable PSA APIs. When this macro is set, the library utilizes PSA Crypto and Secure storage APIs.

Unlike the TF-M project in the TF-M repository, the demo projects in https://github.com/MDK-Packs/TrustZone use precompiled program images for bootloader (See https://github.com/MDK-Packs/TrustZone/tree/main/bl2) and Secure firmware (See https://github.com/MDK-Packs/TrustZone/tree/main/tfm). It is assumed that the bootloader and secure firmware are programmed on the device separately from the Non-secure application program image. As a result, the Non-secure application image needs to be signed as a separate step. This is carried out in the project compilation flow automatically using a User Command setting- as shown in Figure 17:

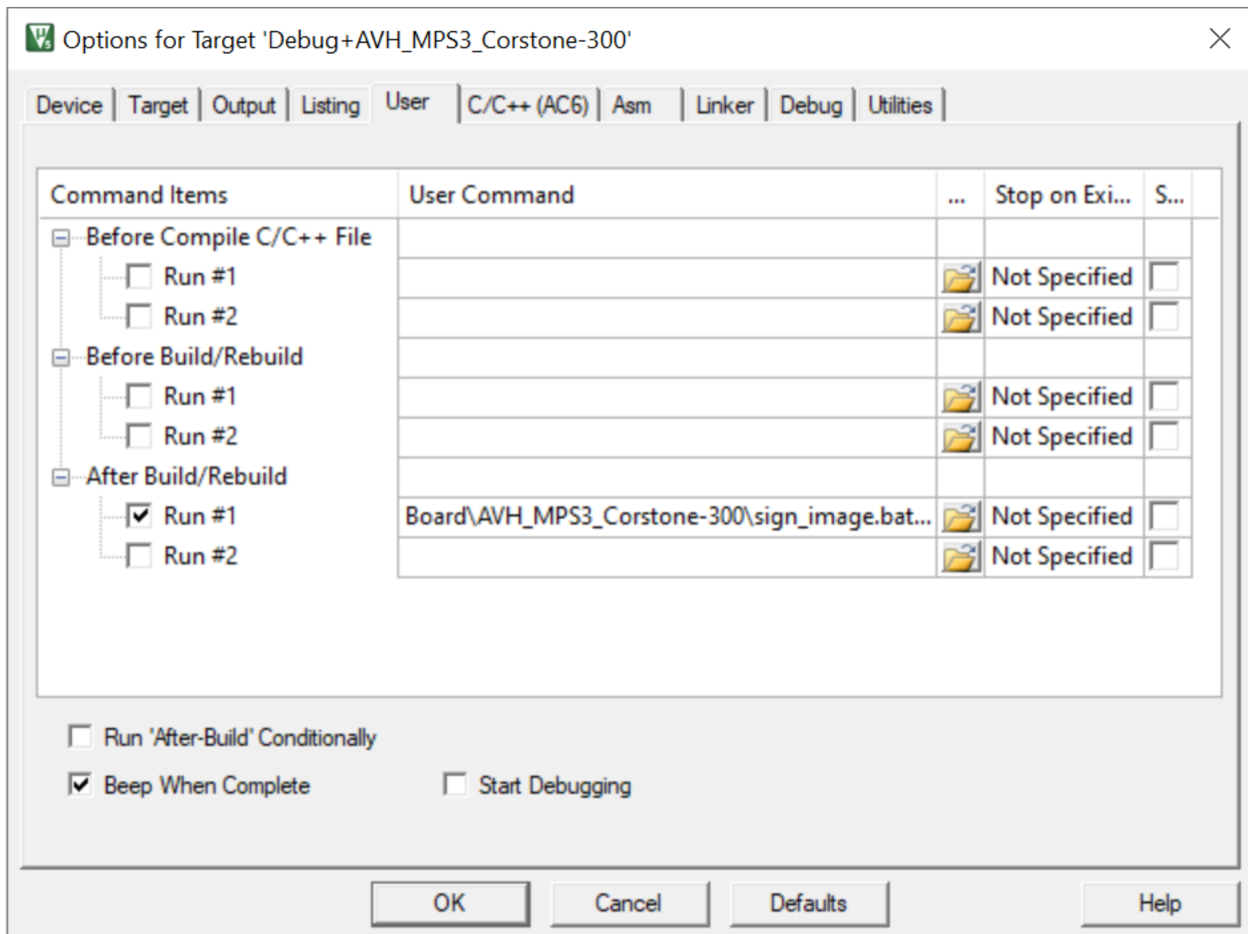

Figure 17: MDK user command option can be used to sign a program image at the end of the compilation.

The full command for signing the image is as follows:

```
Board\AVH_MPS3_Corstone-300\sign_image.bat $L@L "0.9.2" 1
```

(Note: "$L@L" is a built-in symbol in Keil MDK that points to the Linker output file. Additional information on this can be found at https://developer.arm.com/documentation/101407/0537/Utilities/Key-Sequence-for-Tool-Parameters)

"sign_image.bat" is a batch file that can be found at https://github.com/MDK-Packs/TrustZone/blob/main/app/AWS/Board/AVH_MPS3_Corstone-300/sign_image.bat

"sign_image.bat" is a wrapper for imgtool (the same Python module that was mentioned for image signing in the last section). It has a command syntax of: sign_image.bat <name> <version> <counter>

The input parameters are:

- <name>.hex : hex imaged to be signed
- <version> : Version string (e.g. "0.9.2")

- <counter> : security counter (e.g. 1)

The outputs are:

- <name>_signed.bin: Signed binary image (confirmed)
- <name>_OTA.bin: Signed binary image for OTA (not confirmed)
- <name>_signed.hex : Signed hex image generated from <name>_signed.bin

After the image is signed, it can then be programmed into the device and tested.

6 Getting your TF-M setup ready for products

6.1 Configuration options

For product development, a number of configuration options might need to be changed. To do this, you can either:

- edit the configuration files in config subdirectory, or
- override some of them using command line options.

For example, in config/config_base.cmake, you might want to edit the following options:

Options	Note
TFM_PXN_ENABLE	Privileged eXecute Never (PXN) is a memory attribute in the MPU and can prevent unprivileged code from running at privileged level (privilege escalation attacks). This feature was introduced in Armv8.1-M processors (e.g. Cortex-M55, Cortex-M85), and is not available in Armv6-M, Armv7-M and Armv8.0-M. This feature is relevant to isolation levels 2 and 3, but not level 1 as it does not require an MPU.
CONFIG_TFM_HALT_ON_CORE_PANIC	For debugging: you can set this to ON so that the processor halts when there is a fatal error. For production, this option should be set to OFF.
PLATFORM_PSA_ADAC_SECURE_DEBUG	When this is set to TRUE, it enables PSA Authenticated Debug Access Control (ADAC) support in TF-M.
PLATFORM_DEFAULT_xxx	A number of options relating to default keys and non-volatile storage need to be set up. The options are platform dependent.

Also, in config/cp_config_default.cmake

Options	Note
CONFIG_TFM_ENABLE_FP, CONFIG_TFM_ENABLE_MVE_FP	By default, secure firmware does not use the FPU or Helium (MVE) because most of the secure code does not contain floating-point operations. However, if additional secure libraries are added, and if they require floating-point support, this should be set to ON.
CONFIG_TFM_ENABLE_MVE	Enables the integer portion of Helium (MVE) in secure firmware.

CONFIG_TFM_FLOAT_ABI	This specifies the floating-point ABI. If FPU/Helium is not used, then the default is “soft” (i.e. use software to emulate floating-point operations). If FPU/Helium is used, then this should be set to “hard”.

There are additional configuration files which might need to be modified:

Filename	Description
tfm_fwu_config.cmake	Firmware update configuration
tfm_build_log_config.cmake	Configuration of build logs (message output during build)

6.2 Secure boot keys

By default, the TF-M repository comes with example keys for secure boot(s). For production, those keys need to be replaced by your own keys.

6.2.1 BL1 bootloader

The BL1 code folder contains a dummy key pair in the following location:

https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/bl1/bl1_2

If a designer adds BL1 bootloader to their design, the dummy key pair should be replaced. This could be generated using pyhsslms (https://github.com/russhousley/pyhsslms) or similar tools that support LMS key generation.

To use pyhsslms to generate key pairs:

1) Install pyhsslms Python package. For example:

```
$> pip install pyhsslms
```

2) Execute Python and invoke pyhsslms’s genkey function:

```
$> python3 -c "import pyhsslms; priv_key=pyhsslms.HssLmsPrivateKey.genkey('mykey', levels=1,
lms_type=pyhsslms.lms_sha256_m32_h10, lmots_type=pyhsslms.lmots_sha256_n32_w8)"
```

This should generate mykey.prv (private key) and mykey.pub (public key).

6.2.2 BL2 bootloader

By default, MCU boot uses RSA-3072 asymmetric crypto. The default key pairs for this crypto in TF-M is located in the following locations:

- Private key: https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/bl2/ext/mcuboot/root-RSA-3072.pem
- Public key (part of the program code): https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/bl2/ext/mcuboot/keys.c

To generate a key pair, you use the following steps (test.pem is the filename and can be changed):

1) Generate private key : imgtool.py keygen -k test.pem -t rsa-3072
2) Generate public key from private key : imgtool.py getpub -k test.pem

The second step outputs a C char array and you can copy and paste this in keys.c to replace the default key.

6.3 TF-M Built-in Keys

TF-M, in particular the crypto service, provides support for built-in keys. These keys are provisioned in devices (e.g. microcontrollers) and can be used for crypto operations through the TF-M Crypto services by entities (e.g. software components) in both the Secure and Non-secure worlds. However, these built-in keys can only be used via the particular set of key handles associated to them, and are not managed by applications or by other secure services. Examples of these keys are:

- HUK (Hardware Unique Key) – this is normally used by key derivation functions to generate other keys.
- IAK (Initial Attestation Key) – this is used during initial attestation.

Further information on the TF-M built-in keys support and related documentation is available at the following link:

https://tf-m-user-guide.trustedfirmware.org/design_docs/tfm_builtin_keys.html

If the C macro PLATFORM_DEFAULT_CRYPTO_KEYS is defined, then the default key file is used (See https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/interface/include/crypto_keys/tfm_builtin_key_ids.h)

6.4 Default dummy provisioning

By default, the TF-M project is built with dummy keys for provisioning as described in section 6.3. After changing the default keys, we need to set the TFM_DUMMY_PROVISIONING value in config/config_default.cmake so that real hardware provisioning can be carried out. Information appertaining to this is covered in the following page:

https://tf-m-user-guide.trustedfirmware.org/integration_guide/platform_provisioning.html

6.5 Debug vs release

You might notice that the illustrated build command has a build type option of "-DCMAKE_BUILD_TYPE=Debug". This is one of the available options, with the TF-M build system supporting the following build type choices:

- Debug
- RelWithDebInfo (Release with Debug Information)
- Release
- MinSizeRel (Minimum Size Release – this is the default)

Debug symbols are added by default to all build types but can be removed from Release and MinSizeRel builds by setting TFM_DEBUG_SYMBOLS to OFF. These options are covered in the following webpage:

https://tf-m-user-guide.trustedfirmware.org/building/tfm_build_instruction.html?#build-type

7 Other useful information

7.1 Porting to new hardware

Guidelines for porting TF-M to new hardware can be found in the following web pages:

https://tf-m-user-guide.trustedfirmware.org/integration_guide/platform/porting_TFM_to_a_new_hardware.html

https://tf-m-user-guide.trustedfirmware.org/integration_guide/index.html

In addition, you can use an existing port as reference. The following webpages detail the latest hardware support:

- https://tf-m-user-guide.trustedfirmware.org/platform/index.html
- https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/platform/ext/target

7.2 Extending secure firmware

TF-M allows software developers to include additional partitions for Application Root-of-Trust in their secure firmware. This is optional; the majority of simple IoT devices do not need custom defined Application Root-of-Trust. But, in case there is a need to do so, information regarding this can be found here:

https://tf-m-user-guide.trustedfirmware.org/integration_guide/services/tfm_secure_partition_addition.html

It is also possible to have custom defined Secure interrupts. This is documented in the following web page:

https://tf-m-user-guide.trustedfirmware.org/integration_guide/tfm_secure_irq_integration_guide.html

7.3 TF-M Version 1.7

In TF-M version 1.7, there is a number of changes compared to the previous releases. These are:

- Removal of the Library model – The legacy Library model is superseded by the Secure Function Model (SFN).
- Default configuration is replaced with base configuration: In previous versions, the default configuration (when no profile is specified) enables all features. From version 1.7, this is replaced by base configuration and only contains the firmware framework. This means that software developers can enable features based on their project's requirements.
- Configuration mechanism. In previous versions, many of the TF-M configurations were determined by CMAKE variable. From version 1.7, some of these options have moved into C marcos in a C header file (config_base.h).
- Adding Kconfig as a configuration method.
- Adding a new profile i.e. Medium without Application Root-of-Trust (Medium ARoT-less)

TF-M version 1.7 was released in Dec-2022. You can find the release note for version 1.7 at: https://tf-m-user-guide.trustedfirmware.org/releases/1.7.0.html. For additional information about the TF-M roadmap, please visit https://trustedfirmware-m.readthedocs.io/en/latest/roadmap.html.

7.4 Working with the Trusted Firmware team

To get regular updates on Trusted Firmware, you can subscribe to the mailing list. Please visit this page for details: https://www.trustedfirmware.org/contact/

The Trusted Firmware-M team have regular technical meetings. The meeting schedules and agendas are available through the mailing list. Recordings and slides of previous meetings can be found here: https://www.trustedfirmware.org/meetings/tf-m-technical-forum/

As in many open-source projects, the Trusted Firmware team always welcome contributors. Please visit the following webpage for additional information: https://tf-m-user-guide.trustedfirmware.org/contributing/contributing_process.html

7.5 Additional reading

The following table provides the location of key resources and a range of reference materials which could be useful:

Document / website
Trusted Firmware-M Getting Started Guide https://tf-m-user-guide.trustedfirmware.org/getting_started/index.html
Trusted Firmware-M git repository https://git.trustedfirmware.org/TF-M/trusted-firmware-m.git/tree/
Armv8-M Architecture Technical Overview (whitepaper) https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/whitepaper-armv8-m-architecture-technical-overview
Designing Secure IoT Devices with the Arm Platform Security Architecture and Cortex-M33 Author: Trevor Martin Publisher: Newnes ISBN: 978-0-12-821469-5
MCUboot home page https://www.mcuboot.com/
MCUboot github https://github.com/mcu-tools/mcuboot
MCUboot Walkthrough and Porting Guide https://interrupt.memfault.com/blog/mcuboot-overview
Imgtool https://docs.mcuboot.com/imgtool.html
Trusted Firmware-M future plan

https://trustedfirmware-m.readthedocs.io/en/latest/roadmap.html
(Roadmaps for other Trusted Firmware projects can be found in https://trusted-firmware-docs.readthedocs.io/en/latest/general_information/trusted_firmware_roadmaps.html)

Definitive Guide to Arm Cortex-M23 and Cortex-M33 Processors
Author: Joseph Yiu
Publisher: Newnes
ISBN: 978-0-128207352