
User Guide

Release v2.2.1

ARM CE-OSS

Aug 28, 2025

OVERVIEW

1 License	3
2 Folder Structure	5
3 Building examples	7
4 Contribute to this repository	9
4.1 Additional requirements	9
Bibliography	51

The Trusted Firmware-M (TF-M) Extras repository is the extension of the TF-M main repository to host the examples, demonstrations, third-party modules, third-party secure partitions, etc.

LICENSE

The default license of new source code in this repository is [BSD 3-clause](#).

Some source files are contributed by the third-parties or derived from the external projects. A license file should be included in the root folder of these source files if it has a different license.

FOLDER STRUCTURE

- *examples*: This folder hosts the examples and demos for TF-M.
- *partitions*: This folder hosts the supplementary or third-party secure partitions for TF-M.

BUILDING EXAMPLES

Folders represented in the example:

<TF-M Source Dir>	Full path to TF-M Source directory/repository
└─ <toolchain-file-from-tf-m>	
<build-dir>	Common build directory
└─ spe	Secure side build directory
└─ api_ns	Non-Secure API callables directory
└─ nspe	Non-Secure side build directory

1. Building the Secure side (comes from TF-M):

```
$ cmake -S <TF-M Source Dir> \  
        -B <build-dir>/spe \  
        -DTFM_PLATFORM=<tf-m-target> \  
        -DTFM_TOOLCHAIN_FILE=<toolchain-file-from-tf-m> \  
        -DCMAKE_BUILD_TYPE=<Debug,Release>  
$ cmake --build <build-dir>/spe -- -j$(nproc) install
```

2. Build the Non-Secure side (from examples found here):

```
$ cmake -S <example-path> \  
        -B <build-dir> \  
        -DCONFIG_SPE_PATH=<build-dir>/spe/api_ns  
$ cmake --build <build-dir> -- -j$(nproc)
```

Some examples use different flags in Secure and Non-Secure side. Check the examples description for further steps on proper building.

CONTRIBUTE TO THIS REPOSITORY

Refer to [contributing process](#) for the TF-M general contribution guideline.

Please contact [TF-M development mailing list](#) for any question.

Note: If your contribution consists of pre-built binaries, please upload your binary components to [Trusted Firmware binary repository \(tf-binaries\)](#). This repository accepts source files only.

4.1 Additional requirements

- It is expected and strongly recommended to integrate and test your example/secure partition with TF-M latest release, to enable new features and mitigate known security vulnerabilities.
- List the example and secure partition in example readme and secure partition readme respectively.
 - Each example/secure partition shall specify the following information
 - * A brief description
 - * Maintainers with their contact emails
 - * If the example/secure partition is not integrated or tested with the latest TF-M release, specify the TF-M version/commit ID tested with.
 - Each example/secure partition shall follow the structure below

```
Folder name
=====

Description
-----
Simple description

Maintainers
-----
Maintainer list and emails

TF-M version
-----
Optional. Specify the TF-M version/commit ID if it is not integrated or
test with latest TF-M release.
```

4.1.1 Partitions

ADAC implementation for RSE platform

ADAC Requirements for RSE

For RSE, ADAC design and implementation must meet below requirements.

1. Since RSE is HES (Hardware Enforced Security) host for CCA (Confidential Compute Architecture) system, ADAC functionality must be implemented by RSE.
2. By default, CCA HES and other trusted subsystems debug should be disabled all the time.
3. When in a secured (trustworthy) state, no debug should be allowed to RSE, and other components of CCA System security Domain.
4. If life cycle is not in a secured state and if a CCA component debug is requested, a new debug session should be initiated.
5. Likewise at the end of debug session, all debug interfaces should be closed and a system reset is required to return to the previous state.
6. Depending on current policy, the debug start and stop request may require a system reset for the request to be processed in a distinct debug session. For RSE, a system reset is required for handling debug requests for any components of CCA security domain.
7. Finally, CCA Platform Attestation token should be different if any CCA debug is enabled.

Implementation Constraints

PSA ADAC protocol specifies use of asymmetric key cryptography for certificate parsing and authentication. Ideally, authentication and application of permissions should be done at the same time in boot so that they cannot be tampered later on, but

- BL1 is constrained on memory resources and
- BL1 is immutable, so any flaw in the authentication scheme would result in a permanent security vulnerability.

Hence, authentication has to be handled as runtime service while appropriate permissions can be applied in the bootloader.

Design description

As per the ADAC architecture, debug host must implement Secure Debug Manager (SDM) component while debug target requires Secure Debug Authenticator (SDA) as mentioned in architecture specification. Logical link is established among the above two components to establish secure debug connection.

To meet the above requirements, ADAC protocol is integrated in TF-M as follows:

1. A new ADAC runtime service which calls SDA to authenticate any incoming debug request from other components.
2. Above service only acknowledges any incoming debug request if the device is in appropriate life cycle state. Else, it rejects any incoming debug request. Here the appropriate life cycle state is defined by the platform specific policy.
3. Once the service acknowledges the request, it sends the request to the core protocol API for authentication. It also checks if the host has appropriate access rights permissions. If it authenticates the host successfully, it stores the debug state and may initiate the reset (depending on platform policy).

4. On immediate reset, the bootloader (BL1_2) retrieves the stored debug state and applies corresponding debug permissions.
5. It also locks the related DCU bits so that the applied permissions stays the same throughout the debug session.
6. Runtime service now waits for debug end signal to end debug session. To end current debug session, it stores the state again and initiates the reset (depending on platform policy).
7. On reset, BL1_2 resets the permission and locks the DCU to continue normal execution.
8. For debug request of any components where platform policy does not require a reset, ADAC service does not initiate any reset and enables the debug immediately.

Code structure & Service Integration

The ADAC Service source and header files are located in the current directory. The interface for the ADAC runtime Service is located in `interface/include`. The only header to be included by applications that want to use functions from the PSA API is `tfm_adac_api.h`.

Service interface

The ADAC Service exposes the following interface:

```

/*!
 * \brief Authenticates the requested debug service.
 *
 * \param[in] debug_request Request identifier for the debug zone
 *                  (valid values vary based on the platform
 *                  Each bit of the \p debug_request represents
 *                  debug request for corresponding zone.
 *                  e.g.
 *                  If no bits are set => no debug request
 *                  If bit0 is set    => start debug for zone1
 *                  If bit0 is cleared => end debug for zone1
 *                  If bit1 is set    => start debug for zone2
 *                  If bit1 is cleared => end debug for zone2
 *                  ...
 *
 *                  Enumeration of zones (zone1, zone2, etc.) is
 *                  done by ``tfm_debug_zones`` (platform specific)
 *
 * \return Returns PSA_SUCCESS on success,
 *         otherwise error as specified in \ref psa_status_t
 */
psa_status_t tfm_adac_service(uint32_t debug_request)

```

Service source files

- `tfm_adac_api.c`: Implements the secure API layer to allow other services in the secure domain to request functionalities from the adac service using the PSA API interface.
- `adac_req_mgr.c`: Includes the initialization entry of adac service and handles adac service requests in IPC model.
- `adac.c`: Implements core functionalities such as implementation of APIs, handling and processing of debug request.

Hardware abstraction layer Interface

Classification of various debug zones is platform/system specific. For system with RSE subsystem, these are mainly classified into CCA security domain debug and Non-CCA debug zones.

- `tfm_debug_zones`: enumerates 2 CCA and 4 Non-CCA debug zones.
- `tfm_platform_system_reset()`: Request system reset to initiate or terminate a debug session.
- `tfm_plat_otp_read()`: Reads the life cycle state as well as secure debug key required for authentication.

Bootloader Interface

The ADAC runtime service requires to convey debug state information between runtime service and bootloader. This needs be in platform specific predefined persistent area as this information needs to be retained after reset.

For RSE platform, this functionality is provided by RESET_SYNDROME register. 8 bits field, SWSYN, of above register is allocated to convey debug state information between bootloader and runtime service

- **`lcm_dcu_set_enabled()`: Apply appropriate debug zone permissions by setting the DCU register values.**
- **`lcm_dcu_set_locked()`: Locks the DCU so permission cannot be modified during that power cycle.**

ADAC Protocol (SDA) integration

- `tfm_to_psa_adac_rse_secure_debug()`: Initiates the connection with the host debugger and performs secure debug authentication process.

Enable Secure Debug

To enable ADAC on RSE, below options must be configured:

- `-DPLATFORM_PSA_ADAC_SECURE_DEBUG=ON`
- `-DTFM_PARTITION_ADAC=ON`

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

Delegated Attestation Service Integration Guide

Introduction

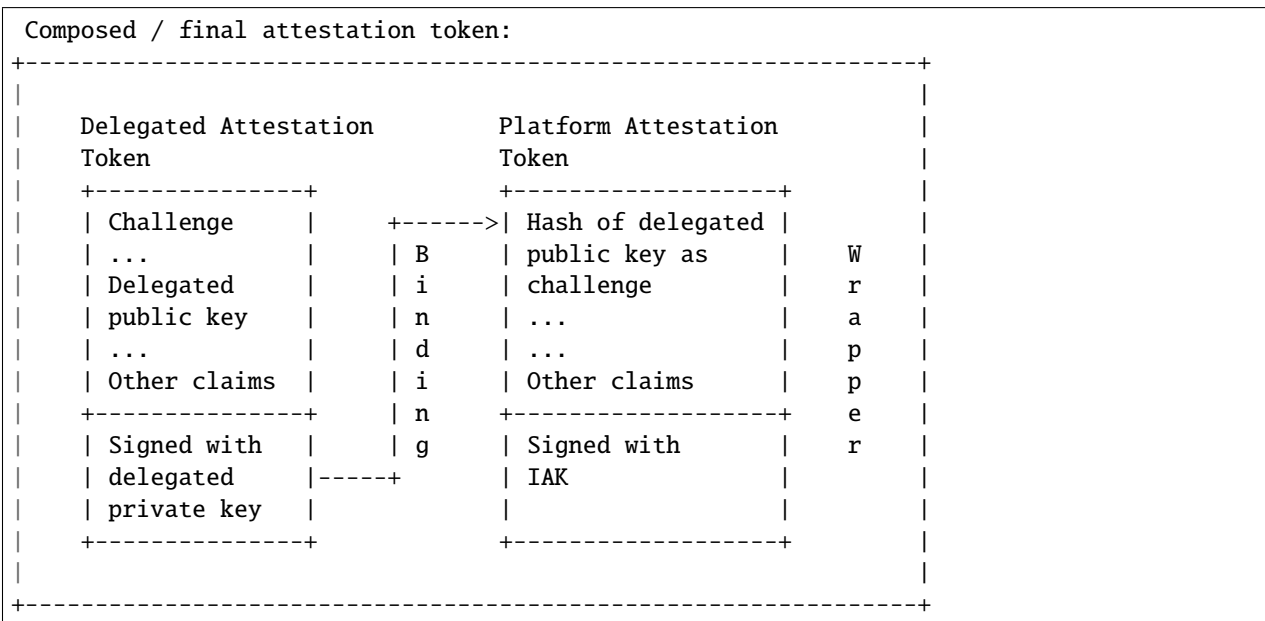
Delegated Attestation Service was mainly developed to support the attestation flow on the ARM Confidential Compute Architecture (ARM CCA)¹. However, it implements a generic model, which fits to other use cases as well. Delegated attestation is a model where the responsibility of creating the overall attestation token is split between different parties. The overall token is a composition of sub-tokens, where each sub-token is produced by an individual entity within the system. Each sub-token is signed with a different key, which is owned by the sub-token producer. The signing keys are derived in a chain. Each key is derived by the producer of the previous (in the chain) attestation token. The sub-tokens must be cryptographically bound to each other, to make the key chain back traceable to the initial attestation key (IAK), which is used to sign the initial attestation token. The cryptographic binding is achieved by including the hash of the public key in the challenge claim of the predecessor attestation token. The IAK or seed of it is provisioned at chip manufacturing time. The rest of the signing keys in the chain are derived at runtime. The main functionalities of the delegated attestation service are:

- Provide an API to derive a delegated attestation key.
- Provide an API to make the previous attestation token (e.g.: initial or platform token) queryable.

The entity at the end of the chain is responsible to compose the final attestation token. This can be achieved by nesting the tokens or adding a wrapper around the sub-tokens.

Usage example:

- Initial or platform attestation token is produced by the Initial Attestation service. Signed by IAK.
- Additional attestation token can be produced by any entity in the system. An entity can request a signing key via the `tfm_delegated_attest_get_delegated_key` call. The previous attestation token can be queried via the `tfm_delegated_attest_get_token` call. The input is the hash of the corresponding public key, which is included as the challenge claim.

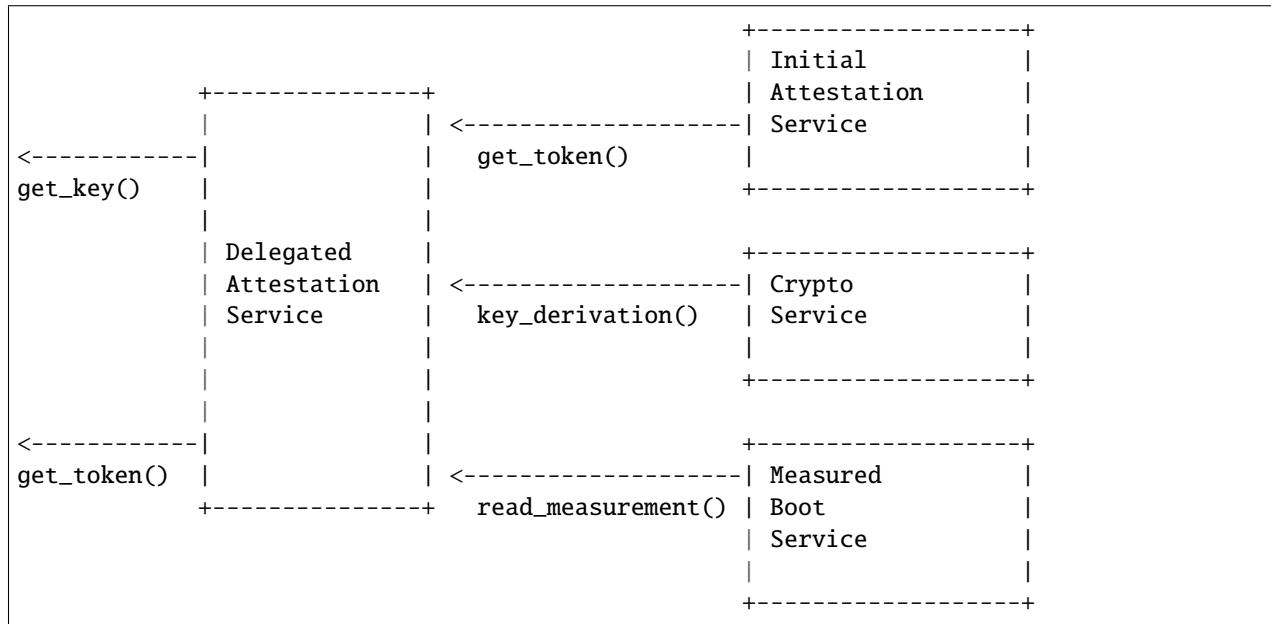


¹ https://developer.arm.com/documentation/DEN0096/A_a/?lang=en

Dependencies

Delegated Attestation service has the following dependencies on other secure services within TF-M:

- Crypto service: Crypto service has access to the pre-provisioned built-in keys. It derives a signing key on request.
- Initial Attestation service: Provides the initial attestation token.
- Measured boot: Provides the firmware measurements and associated metadata. They are used to compute the input for the key derivation.



Delegated Attestation flow diagram

Key derivation

The system **MUST** have a pre-provisioned seed programmed at chip manufacturing time. Several keys could be derived from this seed. The key derivation happens in two phases:

- Boot phase: Done by the bootloader, BL1_1 on RSE platform. The input is the device lifecycle, hash of BL1_2 bootloader, etc.
- Runtime phase: Delegated attestation and crypto services are responsible for the derivation. Delegated attestation computes the inputs from firmware measurements and crypto service does the actual derivation.

Always the same key is derived until the firmware images (and thereby their measurement - hash value is constant) are not changed. If any firmware image gets updated or the device lifecycle has changed, then it will result in a different delegated signing key than the previous one.

Code structure

The TF-M Delegated Attestation Service source and header files are located in the current directory. The interfaces for the delegated attestation service are located in the `interface/include`. The headers to be included by applications that want to use functions from the API is `tfm_delegated_attestation.h` and `tfm_delegated_attest_defs.h`.

Service source files

- `delegated_attest.c` : Implements core functionalities such as implementation of APIs. Interacts with dependent services to derive the signing key and get the initial attestation token.
- `tfm_delegated_attestation_api.c`: Implements the secure API layer to allow other services in the secure domain to request functionalities from the delegated attestation service using the PSA API interface.
- `delegated_attest_req_mgr.c`: Includes the initialization entry of delegated attestation service and handles service requests in IPC model.

Delegated Attestation Interfaces

The TF-M Delegated Attestation service exposes the following interfaces:

```

psa_status_t
tfm_delegated_attest_get_delegated_key(psa_ecc_family_t ecc_curve,
                                       uint32_t      key_bits,
                                       uint8_t      *key_buf,
                                       size_t       key_buf_size,
                                       size_t       *key_size,
                                       psa_algorithm_t hash_algo);

psa_status_t
tfm_delegated_attest_get_token(const uint8_t *dak_pub_hash,
                              size_t       dak_pub_hash_size,
                              uint8_t     *token_buf,
                              size_t       token_buf_size,
                              size_t       *token_size);

```

Related compile time options for out of tree build

- `TFM_PARTITION_DELEGATED_ATTESTATION`: To include the delegated attestation secure partition and its services, its value should be ON. By default, it is switched OFF.
- `TFM_EXTRA_MANIFEST_LIST_FILES`: `<tf-m-extras-repo>/partitions/delegated_attestation/delegated_attestation_manifest_list.ya`
- `TFM_EXTRA_PARTITION_PATHS`: `<tf-m-extras-repo>/partitions/ delegated_attestation`
- `DELEG_ATTEST_DUMP_TOKEN_AND_KEY`: If turned ON then the derived delegated attestation key and the delegated attestation token is printed to the console.

Verification

Regression test

Regression test suite is implemented in `test/delegated_attest_test.c`.

References

Copyright (c) 2022, Arm Limited. All rights reserved.

DICE Protection Environment

The DICE Protection Environment (DPE) service makes it possible to execute DICE commands within an isolated execution environment. It provides clients with an interface to send DICE commands, encoded as CBOR objects, that act on opaque context handles. The DPE service performs DICE derivations and certification on its internal contexts, without exposing the DICE secrets (private keys and CDIs) outside of the isolated execution environment.

For a full description of DPE, see the [DPE Specification](#).

A high level example of DPE commands usage is shown in the below diagram:

DPE consists of both a runtime service and boot time integration. The DPE service is currently a work in progress.

Boot time

A platform integrating DPE must perform the following boot-time functions:

- Derive a RoT CDI from the UDS (HUK) provisioned in OTP, lifecycle state and measurement of the first firmware stage after ROM (BL1_2), and store it via a platform-specific mechanism to be retrieved at runtime.
- Store boot measurements and metadata for all images loaded by the bootloaders in the TF-M shared boot data area.

Runtime DPE service

The runtime DPE service provides the following functionality.

Initialization

At initialization, DPE completes the following tasks:

- Retrieves and processes offline measurements and metadata from the TF-M shared boot data area.
- Retrieves the RoT CDI generated at boot time by calling the `dpe_plat_get_rot_cdi()` platform function.
- Derives DICE contexts for the RoT certificate and platform certificate, using the values processed from boot data and the RoT CDI.
- Shares the initial context handle, corresponding to the newly-created child context, with the first client (AP BL1), via a platform-specific mechanism.

Context management

The internal DICE contexts are referred to by clients of the DPE service using opaque context handles. Each DPE command generates a new context handle that is returned to the client to refer to the new internal state. Each context handle can only be used once, so clients must use the “retain context” parameter of the DPE commands if they wish to obtain a fresh handle to the same context.

The context handles are 32-bit integers, where the lower 16-bits is the index of the context within the service and the upper 16-bits is a random nonce.

The internal contexts are associated with the 32-bit ID of the owner of the context. The DPE service only permits the owner to access the context through its context handle. In the TF-M integration, the ID is bound to the PSA Client ID of the sender of the DPE message.

Client APIs

The DPE partition in TF-M wraps the DPE commands into PSA messages. The request manager abstracts PSA message handling, and the remainder of the service avoids coupling to TF-M partition specifics.

The DPE commands themselves are CBOR-encoded objects that the DPE decode layer decodes into calls to one of the following supported DICE functions.

DeriveContext

Adds a component context to the certificate context, consisting of:

- Context handle
- Parent context handle
- Linked certificate context
- Is leaf
- Client ID
- DICE input values
 - Code hash
 - Config value
 - Authority hash
 - Operating mode

When a certificate context is finalized (`create_certificate=true`), it:

- Computes the Attestation CDI and Sealing CDI.
- Derives an attestation keypair from the Attestation CDI.
- Creates the corresponding certificate and signs it with the previous certificate’s attestation private key.
- Stores the finalized certificate in DPE partition SRAM.

Certificates are created in the CBOR Web Token (CWT) format, using the QCBOR and `t_cose` libraries. CWT is specified in [RFC 8392](#), with customization from [Open DICE](#).

DeriveContext flow diagram

GetCertificateChain

Returns the full certificate chain leading to a given context.

- Returns the certificate chain (collection of individual certificates) as a CBOR array with format [+COSE_Sign1, COSE_Key]. The (pre-provisioned) root attestation public key is the first element in the CBOR array.

The following diagram shows a example certificate chain for RSE TC platform:

CertifyKey

Generates and returns a leaf certificate. If a public key is supplied, then it certifies the key. If a public key is not supplied, then it derives key pair from the accumulated context information for that certificate and certifies the public key.

- If the input certificate context (certificate linked to component context) is already finalised, then it creates a new leaf certificate with no measurements.
- If the input certificate context is not finalised, then it creates a leaf certificate with all the measurements accumulated for that certificate context.
- Adds label (if supplied) to list of measurements.
- Returns the leaf certificate.

Seal

Encrypts and authenticates data using two keys derived from the Sealing CDI, identifiers of the software components in the chain and a supplied label.

- Not currently implemented.

Unseal

Inverse of Seal.

- Not currently implemented.

Host Build

Tested only on Linux and RSE as build platform.

To enable the host build add this to the regular CMake command line:

```
-DHOST_BUILD=ON
```

Example script (tf-m, tf-m-tests, tf-m-extras path need to be updated):

```
#!/usr/bin/env bash
## Update this part ###
TFM_PATH=<tf-m path>
TFM_TEST_PATH=<tf-m-tests path>
TFM_EXTRAS_PATH=<tf-m-extras path>
CROSS_COMPILER_PATH=<cross-compiler path>
# Create the build directory
cd $TFM
rm -rf build
mkdir build
# Execute CMake configuration step to generate the build files
cmake \
-S $TFM_PATH \
-B $TFM_PATH/build \
-DTFM_PLATFORM=arm/rse/tc/tc2 \
-DTFM_TOOLCHAIN_FILE=$TFM_PATH/toolchain_GNUARM.cmake \
-DCMAKE_BUILD_TYPE=Debug \
-DMCUBOOT_IMAGE_NUMBER=4 \
-DRSS_GPT_SUPPORT=0 \
-DTFM_EXTRAS_REPO_PATH=$TFM_EXTRAS_PATH \
-DTFM_SPM_LOG_LEVEL=3 \
-DRSE_LOAD_NS_IMAGE=OFF \
-DTFM_ISOLATION_LEVEL=1 \
-DCONFIG_TFM_SPM_BACKEND=IPC \
-DTFM_TEST_PATH=$TFM_TEST_PATH \
-DCROSS_COMPILE=$CROSS_COMPILER_PATH/gcc-arm-11.2-2022.02-x86_64-arm-none-eabi/bin/arm-
none-eabi \
-DHOST_BUILD=ON
# Go to the build folder to execute only a partial build
cd $TFM_PATH/build
# Build only the host_app target and skip the rest
make dpe_host
```

The compiled dpe_host app is installed to here:

```
<TFM_PATH>/build/bin/host/dpe
```

There are two main operational modes of the dpe_host app:

- Regression mode: Invoking without any command line parameter results in executing the regular regression test suite.
- Fuzzer mode: Invoking with [-c, -d, -k -g, -r] options can be used to execute a single DPE command.

Code coverage

The code coverage measurement is by default enabled in the DPE host build. The coverage report can be generated as follows:

```
# Find where the *.gcda files are created
cd <TFM_BUILD_DIR>
find . -name *.gcda
lcov --capture --directory <LOCATION_OF_GCDA_FILES> --output-file ./dpe.info
genhtml --output-directory=./dpe_coverage ./dpe.info
```

Fuzz test

Compile and install AFL++,

Read the [doc](#) on how to use the fuzzer.

Create a symlink to afl-cc:

```
sudo ln -s afl-cc afl-clang-lto
```

Export this environment variable to instrument only the relevant part of the code:

```
export AFL_LLVM_ALLOWLIST=<TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/
↪fuzz/allowlist.txt
```

Add this argument to the CMake command in the Host Build section.

```
-DAFL_CC=<OFF, afl-clang-lto, ..>
```

Recompile the dpe_host app with afl-cc.

Execute fuzzing:

```
# Fuzz DeriveContext (dc)
afl-fuzz -i <TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/fuzz/input/raw/
↪dc \
-o <TFM_PATH>/fuzz_out \
-- <TFM_PATH>/build/bin/host/dpe \
-d @@

# Fuzz CertifyKey (ck)
afl-fuzz -i <TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/fuzz/input/raw/
↪ck \
-o <TFM_PATH>/fuzz_out \
-- <TFM_PATH>/build/bin/host/dpe \
-k @@

# Fuzz GetCertificateChain (gcc)
afl-fuzz -i <TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/fuzz/input/raw/
↪gcc \
-o <TFM_PATH>/fuzz_out \
-- <TFM_PATH>/build/bin/host/dpe \
-g @@

# Fuzz CBOR parser (cbor)
afl-fuzz -i <TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/fuzz/input/
↪cbor \
```

(continues on next page)

(continued from previous page)

```
-o <TFM_PATH>/fuzz_out \
-- <TFM_PATH>/build/bin/host/dpe \
-c @@
```

Generate initial input for the fuzzer:

- Raw input means that a simplified subset (buffer related arguments are ignored) of the DPE command arguments are provided in a binary format and functions in `<TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/host/cmd.c` turns those into real DPE commands through the DPE client API calls. As a result, the CBOR encoding of the commands is proper. The goal is to avoid error cases due to CBOR encoding in the command decoder part and be able to test the main functionality of the command. The raw binary input files can be generated based on these hexdump like files: `<TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/fuzz/input/raw/*.txt` with the following commands:

```
xxd -r dc.txt dc_cmd.bin
xxd -r ck.txt ck_cmd.bin
xxd -r gcc.txt gcc_cmd.bin
```

Modifying the content of the ```*.txt``` files and generating the binary files results **in** the modification of the DPE **command** arguments.

The first byte of the raw data is not strictly related to the DPE command. It means to indicate which hard-coded **command** sequence to executes before executing the actual input. These hard-coded **command** sequences can be used to build a certain state (certificate chain) of the service. They can be found here:

```
``<TFM_EXTRAS_PATH>/partitions/dice_protection_environment/test/dpe_test_data.c``
```

DPE **command** arguments are mostly boolean values. To ensure the normal distribution of these **in** the DPE commands, therefore odd value are converted to **true** and even values to **false in** the raw input.

- CBOR input means that the input provided through the command line is already a proper CBOR encoded DPE command. The input does not go through the DPE client API instead it is passed directly to the command parser. When the fuzzer modifies the CBOR input it is expected that a lot of CBOR encoding error will appear in the input. Therefore this is meant to mainly test the command parser part of the DPE service. This type of input is collected so that during the regression test the DPE commands are printed to the console and these are turned into binary files.

Copyright (c) 2023-2024, Arm Limited. All rights reserved.

Readme

DMA-350 privilege separation

The DMA-350 component consists of multiple channels, that can work independently. These channels can be limited in both privilege and security. When Non-secure access is granted to a DMA channel, the device automatically limits the generated transfers to Non-secure. The same happens, when unprivileged access is granted to a channel: the transfers will be limited to unprivileged. However, there is a risk in the latter: If a system does not utilize a system MPU to filter DMA accesses to the memory system, unprivileged DMA channels can access to privileged data, and this results in security issues.

For such systems, if unprivileged tasks need access to the DMA, Arm recommends the following, both for Secure and Non-secure applications:

1. Set all DMA channels to privileged in the DMA controller
2. Grant read-write accesses in MPU for the DMA channel register blocks for the unprivileged task
3. Create a privileged (checker) task, that can be called from an unprivileged task (eg. via SVC call):
 - a. Use this privileged task to provide a limited API towards the unprivileged task
 - b. The privileged task can check the parameters and the addresses that would be accessed by the DMA operation. Based on the result, it can setup the DMA channel with the requested parameters or deny it completely

Note:

- It is assumed that the Operation System provides a method to define MPU configurations as a per task basis, and always updates the MPU configuration for the current running unprivileged task. When the privileged (checker) task is called, the current MPU configuration - which is of the (unprivileged) caller task - should remain intact. This way, when the privileged (checker) task queries the MPU for unprivileged access with an address, the result will be from the perspective of the (unprivileged) caller task.
- The API provided by the privileged task should be limited, as checking complex configuration would require simulating the inner workings of the DMA itself.
- In case dynamic assignment of the channels is required, special care must be taken when reconfiguring the privilege or security of a channel. To prevent information leakage and privilege escalation, reconfiguration of a channel can only happen on an idle channel, and it also results in resetting the registers of that channel.

The following diagram shows an example call flow from Unprivileged Task 1 to configuring the DMA channel:

1. Unprivileged Task 1 calls the DMA-350 unprivileged API
2. DMA-350 unprivileged API function makes an SVC call with an arbitrary, but unique SVC number (this must be different from SVC services that the RTOS already offered.)
3. SVC Handler forwards the call to the DMA-350 specific handler
4. Privileged (checker) task extracts the parameters from the caller stack and query the MPU:
 - a. Requested channel register for read access
 - b. Requested source address range for read access
 - c. Requested destination address range for read-write access
5. If all query pass, configure the requested channel. If not, return an error status and this error information propagate back to the unprivileged task 1.

Some example functionality the unprivileged API can provide:

- Memory copy, memory move, endian swap copy
- Triggers for application specific configurations: The DMA is configured by privileged tasks; API only to start/stop/query the given channel

The following list demonstrates how are unprivileged accesses are checked and blocked.

1. Direct from an unprivileged task: Denied by the built-in check of the DMA. It will result in either security violation or RAZ/WI (read as zero, write ignored), depending on DMA configuration

2. Access through the API, request for a channel that is not dedicated to the Blocked by the privileged (checker) task. When the request is made, the task queries the MPU, whether there is unprivileged read access for the channel registers, using the TTT instructions
3. Access through the API, with requested read/write addresses in privileged region: Blocked by the privileged (checker) task. When the request is made, the task can queries the address range (that would be affected by the operation) in the MPU for unprivileged access, using the TTT instructions or the `cmse_check_address_range` intrinsic
4. Access through the API, with requested read/write addresses in unprivileged channel register block region: Blocked by the target DMA channel. The checker layer will allow the transaction, as in the MPU, the target address is set to unprivileged. The transaction will be generated as an unprivileged transfer, but as the target DMA channel is set to privileged in the DMA controller, it will block the access. It will result in either security violation or RAZ/WI, depending on DMA configuration

Software developers must also consider:

- How to inform the unprivileged task that a DMA operation is completed - The DMA interrupt handle could utilize RTOS event communication to send an event to the unprivileged task to indicate that the DMA operation is completed.
- How errors are handled - If a DMA operation results in an error, potentially the error handler (execute in privileged state) can inform the unprivileged task via RTOS message/event queue.

To support these communications, the unprivileged APIs and the SVC services for setting up DMA-350 might require RTOS specific parameters to define the OS events / message queue to use.

Copyright (c) 2022, Arm Limited. All rights reserved.

TF-M application root of trust partition example for the unprivileged DMA-350 library. It is expected to be used in Isolation Level 2, as the unprivileged API checks the access rights based on the MPU configuration. The example demonstrates the proper, non-blocking usage of the library, as well as some negative tests for invalid channel access, not allocated channel access, and accesses for privileged memory. For detailed description of how privilege separation can be achieved with DMA-350, checkout [DMA-350 privilege separation](#) The partition requires a DMA350 peripheral in the platform with Channel 0 configured as secure, like for example `mps3/corstone310/fvp`.

Build steps for mps3/corstone310/fvp platform

1. Run the following command in the tf-m directory:

```
$ cmake -S . -B cmake_build -DTFM_PLATFORM=arm/mps3/corstone310/fvp -DTFM_TOOLCHAIN_
↪FILE=toolchain_ARMCLANG.cmake -DTFM_ISOLATION_LEVEL=2 -DPLATFORM_SVC_HANDLERS=ON -DTFM_
↪EXTRA_PARTITION_PATHS=<tf-m-extras root>/partitions/dma350_unpriv_partition -DTFM_
↪PARTITION_LOG_LEVEL=TFM_PARTITION_LOG_LEVEL_INFO -DTFM_EXTRA_MANIFEST_LIST_FILES=<tf-m-
↪extras root>/partitions/dma350_unpriv_partition/extra_manifest_list.yaml
```

2. Then:

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

Copyright (c) 2022, Arm Limited. All rights reserved.

External Trusted Secure Storage

Abstract

This document mainly introduces the motivation of adding External Trusted Secure Storage(ETSS) partition and the corresponding design proposal.

Introduction

A secure storage solution is very important when storage is external to MCU. Macronix and other Flash memory suppliers have developed several security memory products, and three major products are RPMC, Authentication Flash, and a more full featured secure Flash like Macronix ArmorFlash.

RPMC is a memory device which provides non-volatile monotonic counters for replay protection.

Authentication Flash mainly provides authentication mechanism to enhance the security of data transmission.

Compared to previous two security Flash, the full featured secure Flash performs authentication, encryption along with a full range of additional security features. This secure Flash generally equips with hardware crypto engine with advanced cryptography algorithms, physically unclonable function(PUF), non-volatile monotonic counters, TRNG, key storage and management module, etc. Secure Flash always provides protection against hardware attacks such as probing, side-attack and fault injection.

In addition, the communication channel between host MCU/SoC and secure Flash is protected by encryption, authentication, data scrambling, and frame sequencing with monotonic counters, as shown in *Secure communication channel between host and secure Flash*. Besides, the independent secure sections configured with specific security policy satisfies multi-tenant isolation.

Hence the secure Flash provides dependable defense against unauthorised access, man-in-the-middle, replay, sniffing and other security threats.

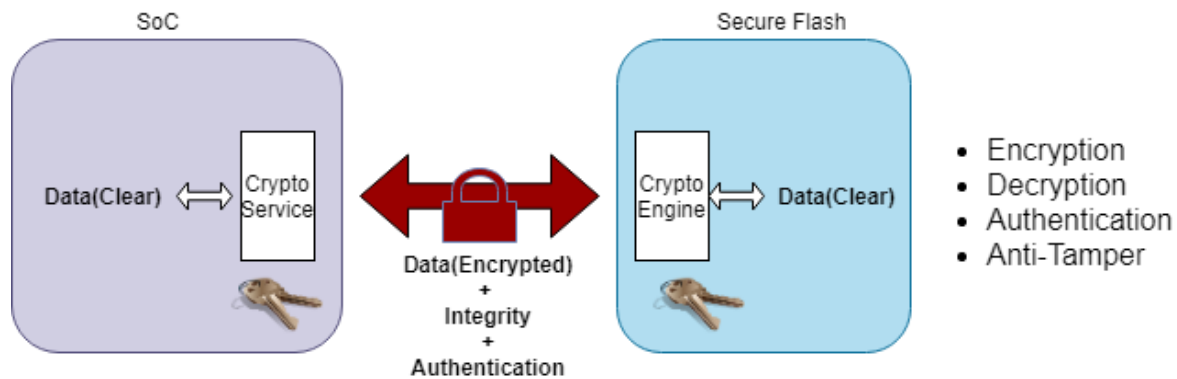


Figure 1:: Secure communication channel between host and secure Flash

More information about secure Flash can be extracted from Macronix ArmorFlash product introduction¹ and the ArmorFlash Whitepaper² for understanding the secure memory architectures in emerging electronic systems.

¹ ArmorFlash product instruction

² ArmorFlash Whitepaper

Design concept

Overview

An ETSS partition is developed as a PSA RoT secure partition to provide external trusted secure storage services based on above external security memory products features. These three major security memory products are collectively referred to as secure Flash in the following. The ETSS partition includes several software components, which are listed below:

Component name	Description
service API	The service interface of ETSS partition to the NSPE/SPE
Service module	This module handles the service calls from NSPE/SPE
Secure Flash framework module	This module is the generic framework of secure Flash driver.

The interaction between these different components is illustrated in the following block diagram:

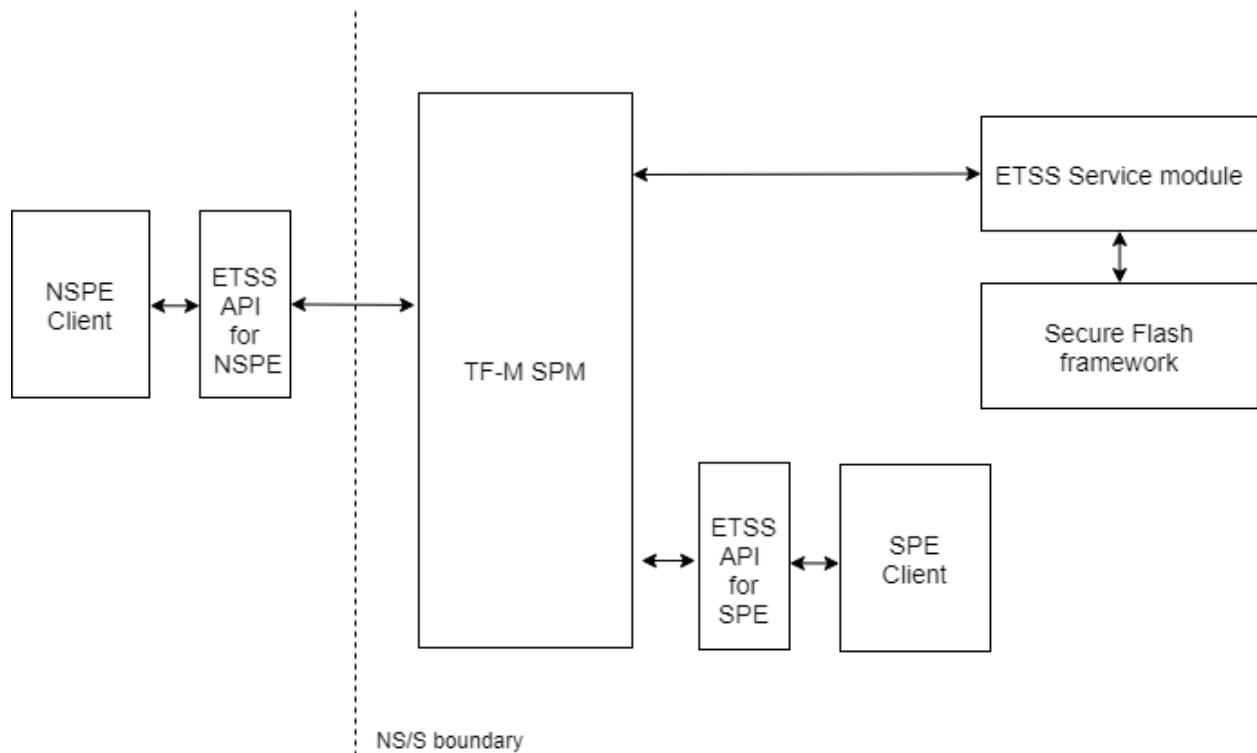


Figure 2:: Block diagram of the different components of ETSS partition.

The more detailed architecture of ETSS service with secure Flash framework is shown below.

ETSS services can be accessed by other services running in SPE, or by applications running in the NSPE.

ETSS services are split into two independent parts: provisioning and deployment. A secure Flash provisioning process should be performed before deployment to set up binding keys and grant access rights. The `etss_secure_flash_provisioning` service is provided to perform secure Flash provisioning in the manufacture process. The specific provisioning implementation may vary with security memory vendors and platforms.

After provisioning, ETSS is ready for providing deployment services with external secure Flash. The available services vary with security memory products. There are three types of services:

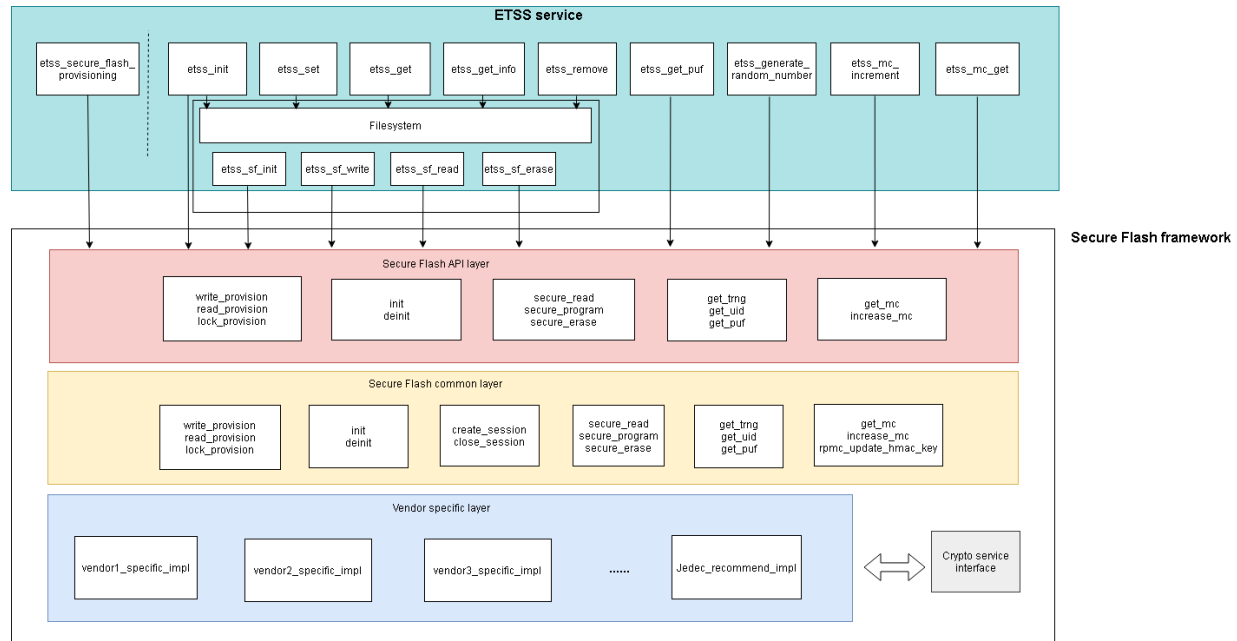


Figure 3:: Layered architecture of ETSS service with secure Flash framework

- Secure storage
- Replay protection monotonic counter manipulation
- Extra services based on extended security features(such as PUF, true random number generator, etc.)

The actually available services are based on the security features of backend secure Flash.

Taking following scenarios for example:

- The external security memory product is just an RPMC, only monotonic counters manipulation services are available.
- The external security memory product is a full featured secure Flash, it supports security read, security program, has a certain number of monotonic counters and other extra security functions. Then the holistic ETSS services may be available.

In the current implementation, ETSS partition just copies the concise filesystem implemented in TF-M ITS partition. As this filesystem doesn't involve access rights management, to support secure Flash multi-zone isolation, it needs to declare separate filesystem contexts for each secure Flash isolated partition. The detailed layout of each isolated partition is set up by the specific secureflash_layout.h of each secure Flash. For each specific security memory products, the secureflash_layout.h should be configured according to the application scenario.

If user needs to support two and more security memory products simultaneously in ETSS partition, then corresponding secure Flash instances and filesystem contexts should be declared.

The secure Flash framework module aims to generalize the application interface of secure Flash driver, and cover different vendors' security memory products. It can be integrated with different software platforms and OSes, and consists of four parts: secure Flash API layer, secure Flash common layer, vendor specific layer and crypto service interface.

- Secure Flash API layer: This layer mainly manages application's access permission based on application identification and pre-provisioned information. The implementation of this layer varies across software platforms and OSes. Here integrated with TF-M, this layer manages access permissions based on client id, and derives parameters passed to secure Flash common layer.

- Secure Flash common layer: This layer abstracts secure Flash operations, and calls binding vendor specific operations.
- Vendor specific layer: The specific implementation of different secure Flash vendors and JEDEC recommended implementation, it depends on upper layer's choice to bind with JEDEC recommended implementation or vendor specific implementation. This layer calls tf-m crypto services via crypto service interface to perform cryptographic operations, then assemble packets sent to external secure Flash and parse packets received from external secure Flash.

If vendors tend to contribute projects with hiding some critical source codes, then these critical parts can be released as library files. These library files may be maintained in another git repository because of different license, vendors should explain how to access these library files in relevant documents.

Code structure

The code structure of this partition is as follows:

tf-m-extras repo:

partitions/external_trusted_secure_storage/etss_partition/

- `etss.yaml` - ETSS partition manifest file
- `etss_secure_api.c` - ETSS API implementation for SPE
- `etss_req_mgr.c` - Uniform IPC request handlers
- `external_trusted_secure_storage.h` - ETSS API with `client_id` parameter
- `external_trusted_secure_storage.c` - ETSS implementation, using `secureflash_fs` as back-end
- `secureflash_fs/` - Secure Flash filesystem
- `external_secure_flash/` - Secure Flash filesystem operations
- **secureflash/ - Backend secure Flash framework for ETSS service**
 - `secureflash.c` - Secure Flash API layer interfaces implementation
 - `secureflash.h` - Secure Flash API layer interfaces
 - `secureflash_common/` - Secure Flash common layer of secure Flash framework
 - `crypto_interface/` - Crypto service interface of secure Flash framework
 - `JEDEC_recommend_impl/` - Reserved JEDEC recommend uniform implementation
 - `macronix/` - Macronix specific implementation
 - `secureflash_vendor2/` - Reserved vendor2 specific implementation
 - `secureflash_vendor3/` - Reserved vendor3 specific implementation
- `template/` - Templates of underlying hardware platform specific implementation of ETSS service

interface/

- `include/etss/etss_api.h` - ETSS API
- `include/etss/etss_defs.h` - ETSS definitions
- `src/etss/etss_ipc_api.c` - ETSS API implementation for NSPE

suites/etss

- `non_secure/etss_ns_interface_testsuite.c` - ETSS non-secure client interface test suite

Future changes

Currently, the implementation of secure Flash provisioning service is primitive, and only suitable for developer mode. In the future, a proper secure Flash provisioning implementation will be provided.

Besides, the following works are underway:

- Optimize secure Flash sessions management.
- Add access rights management features to secure Flash filesystem.

References

Copyright (c) 2021-2022, Macronix International Co. LTD. All rights reserved. SPDX-License-Identifier: BSD-3-Clause

Measured Boot Service Integration Guide

Introduction

Measured Boot partition provides services to extend and read measurements (hash values and metadata) during various stages of a power cycle. These measurements can be extended and read by any application/service (secure or non-secure).

Measurements

The initial attestation token (required by attestation service) is formed of various claims. Each software component claim comprises of the following measurements which are extended and read by Measured Boot services.

- **Measurement type:** It represents the role of the software component. Value is encoded as a short(!) text string.
- **Measurement value:** It represents a hash of the invariant software component in memory at start-up time. The value must be a cryptographic hash of 256 bits or stronger. Value is encoded as a byte string.
- **Version:** It represents the issued software version. Value is encoded as a text string.
- **Signer ID:** It represents the hash of a signing authority public key. Value is encoded as a byte string.
- **Measurement description:** It represents the way in which the measurement value of the software component is computed. Value is encoded as text string containing an abbreviated description (name) of the measurement method.

Code structure

The TF-M Measured Boot Service source and header files are located in current directory. The interfaces for the measured boot service are located in the `interface/include`. The headers to be included by applications that want to use functions from the API is `measured_boot_api.h` and `measured_boot_defs.h`.

Service source files

- **Measured Boot Service:**

- `measured_boot.c` : Implements core functionalities such as implementation of APIs, extension and reading of measurements.
- `measured_boot_api.c`: Implements the secure API layer to allow other services in the secure domain to request functionalities from the measured boot service using the PSA API interface.
- `measured_boot_req_mgr.c`: Includes the initialization entry of measured boot service and handles service requests in IPC model.

Measured Boot Interfaces

The TF-M Measured Boot service exposes the following interfaces:

```

psa_status_t tfm_measured_boot_read_measurement(
    uint8_t index,
    uint8_t *signer_id,
    size_t signer_id_size,
    size_t *signer_id_len,
    uint8_t *version,
    size_t version_size,
    size_t *version_len,
    uint32_t *measurement_algo,
    uint8_t *sw_type,
    size_t sw_type_size,
    size_t *sw_type_len,
    uint8_t *measurement_value,
    size_t measurement_value_size,
    size_t *measurement_value_len,
    bool *is_locked);

psa_status_t tfm_measured_boot_extend_measurement(
    uint8_t index,
    const uint8_t *signer_id,
    size_t signer_id_size,
    const uint8_t *version,
    size_t version_size,
    uint32_t measurement_algo,
    const uint8_t *sw_type,
    size_t sw_type_size,
    const uint8_t *measurement_value,
    size_t measurement_value_size,
    bool lock_measurement);

```

When reading measurement, the caller must allocate large enough buffers to accommodate data for all the output measurement parameters. The definitions `SIGNER_ID_MAX_SIZE`, `VERSION_MAX_SIZE`, `SW_TYPE_MAX_SIZE`, and `MEASUREMENT_VALUE_MAX_SIZE` can be used to determine the required size of the buffers.

System integrators might need to port these interfaces to a custom secure partition manager implementation (SPM). Implementations in TF-M project can be found in `tf-m-extras` repository.

- `partitions/measured_boot/interface/src/measured_boot_api.c`: non-secure as well as secure interface implementation

Related compile time options for out of tree build

- `TFM_PARTITION_MEASURED_BOOT`: To include measured boot secure partition and its services, its value should be ON. By default, it is switched OFF.
- `MEASURED_BOOT_HASH_ALG`: This option selects the hash algorithm used for extension of measurement hashes. Its default value is `PSA_ALG_SHA_256`.
- `TFM_EXTRA_MANIFEST_LIST_FILES`: `<tf-m-extras-repo>/partitions/measured_boot/measured_boot_manifest_list.yaml`
- `TFM_EXTRA_PARTITION_PATHS`: `<tf-m-extras-repo>/partitions/measured_boot`

Verification

Regression test

To be implemented.

Copyright (c) 2022, Arm Limited. All rights reserved.

SCMI Comms Partition

The SCMI Comms partition provides a minimal implementation of the SCMI¹ protocol for the purpose of subscribing to system power state notifications from SCP.

It currently supports only the shared memory based transport protocol.

Supported message types

The partition supports the System power management protocol¹. It can send the following message types:

- `SYSTEM_POWER_STATE_NOTIFY`

It can receive the following message types:

- `SYSTEM_POWER_STATE_SET`
- `SYSTEM_POWER_STATE_NOTIFIER`

Code structure

Partition source files:

- `scmi_comms.c`: Implements the core SCMI message handling.
- `scmi_comms.h`: Common definitions used within the partition.
- `scmi_hal.h`: Hardware abstraction layer that must be implemented by the platform to support the SCMI Comms partition.
- `scmi_protocol.h`: Defines values from the SCMI spec.

¹ Arm System Control and Management Interface (SCMI)

Build options for out of tree build

- `TFM_PARTITION_SCMI_COMMS`: To build the SCMI Comms secure partition its value should be ON. By default, it is switched OFF.
- `TFM_EXTRA_MANIFEST_LIST_FILES`: `<tf-m-extras-repo>/partitions/scmi/scmi_comms_manifest_list.yaml`
- `TFM_EXTRA_PARTITION_PATHS`: `<tf-m-extras-repo>/partitions/scmi`

Platform porting

To use the SCMI Comms partition, a platform must supply an interface library called `scmi_hal` for the partition to link against. The library must contain implementations of all of the functions declared in `scmi_hal.h`. It must also contain a header called `scmi_hal_defs.h`, with the following definitions:

- `SCP_SHARED_MEMORY_BASE`: The base address of a memory area shared between SCP and the CPU running TF-M, which is used to pass messages via the SCMI shared memory transport protocol.
- `SCP_SHARED_MEMORY_SIZE`: The size of the SCP shared memory area. The maximum SCMI message length that can be transported is the size of this area minus the 24 bytes used by the transport protocol.

Additionally, the platform must define `SCP_DOORBELL_IRQ` to be the IRQ number triggered by the SCP doorbell in its `config_tfm_target.h` header. It must also implement that IRQ's handler function to route the request to the SCMI comms partition (see [TF-M Secure IRQ integration guide](#) for more details).

Testing

A regression test suite for the Secure processing environment is provided in `test/secure/scmi_s_testsuite.c`. To test the partition locally, the tests rely on modifying the partition to use the `TFM_TIMER0_IRQ` IRQ source to trigger its interrupt handler. The tests then use the `tfm_plat_test.h` APIs to trigger the timer interrupt and cause the partition to handle an SCMI message. They also reimplement the HAL so that the shared memory and doorbell state are in local memory.

To run the tests, all of the following build options need to be supplied:

- `TFM_EXTRA_MANIFEST_LIST_FILES`: Change to use `<tf-m-extras-repo>/partitions/scmi/test/secure/scmi_comms_manifest_list.yaml` instead of the standard manifest.
- `EXTRA_S_TEST_SUITE_PATH`: `<tfm_extras_dir>/partitions/scmi/test/secure`
- `TEST_S_SCMI_COMMS`: Set to ON to enable the tests and test HAL.

References

SPDX-License-Identifier: BSD-3-Clause

SPDX-FileCopyrightText: Copyright The TrustedFirmware-M Contributors

The list and simple introduction of 3rd-party Secure Partitions in this folder.

DMA-350

DMA-350 Example unprivileged partition

Maintainers

- Bence Balogh bence.balogh@arm.com
- Mark Horvath mark.horvath@arm.com

Measured Boot

Measured boot partition for extending and retrieving software component measurements for RSE platform.

Maintainers

- Maulik Patel Maulik.Patel@arm.com
- David Vincze David.Vincze@arm.com

External Trusted Secure Storage

ETSS partition for providing external trusted secure storage services to protect assets stored in external secure Flash from a variety of security attacks. Available from **TF-M v1.4.0**

Maintainers

- Poppy Wu poppywu@mxic.com.cn

Delegated Attestation

The aim of the partition is to support platforms/systems using a delegated attestation model by providing services for delegated key generation and platform attestation token creation.

Maintainers

- David Vincze David.Vincze@arm.com

Voice Activity Detection

Secure partition for the AN552 FPGA image. It implements voice activity detection on the microphone input of the MPS3 board, and if voice detected (which can be any noise) a short sample (~100 ms) is recorded. Then it can be calculated that which frequency component has the highest energy in the recorded sample.

Maintainers

- Gabor Toth gabor.toth@arm.com
- Mark Horvath mark.horvath@arm.com

ADAC

ADAC partition for authenticated debug and access control for RSE platform.

Maintainers

- Maulik Patel Maulik.Patel@arm.com

Dice Protection Environment

The partition aims to provide DICE command services to create, store and manage DICE secrets.

Maintainers

- Maulik Patel Maulik.Patel@arm.com

SCMI Comms

A partition that can subscribe to SCMI system power state notifications from SCP.

Maintainers

- Jamie Fox jamie.fox@arm.com

Copyright (c) 2021-2024, Arm Limited. All rights reserved.

4.1.2 Examples

Non-Secure DMA350 example for FreeRTOS

DMA350 Triggering interface example

Example usage of triggering flow control with DMA350. The DMA350 is configured to control the data exchange with the UARTs. The CPU can enter into WFI() and the DMA will signal, when the transactions are done. The CPU only wakes up to process the received data, then goes back to sleep.

Build steps

1. Build Secure TF-M with the following commands:

```
$ cmake -S <TF-M Source Dir> -B build/spe -DTFM_PLATFORM=arm/mps3/corstone310/fvp -DTFM_
↪PROFILE=profile_small
$ cmake --build build/spe -- -j$(nproc) install
```

2. Then to build the Non-Secure app:

```
$ cmake -S <path_to_this_example> -B build/nspe -DCONFIG_SPE_PATH=<absolute_path_to>/
↪build/spe/api_ns
$ cmake --build build/nspe -- -j$(nproc)
```

Run steps

The example can run only with 11.22.35 or later versions of Corstone SSE-310 Arm Ecosystem FVP. The `mps3_board. uart1_adapter_tx.ENABLE` and `mps3_board. uart0_adapter_rx.ENABLE` parameters have to be set, to enable the triggering interface of the UARTs. The `mps3_board. uart0.rx_overrun_mode=0` parameter is needed. UART overrun can happen when the received data is not handled in time. The UART overrun interrupt is turned off to prevent lock-up, but there might be data loss when the user sends data during data processing or UART transmitting.

1. Run the following command:

```
./FVP_Corstone_SSE-310 -a cpu0*="bl2.axf" --data "tfm_s_ns_signed.bin"@0x38000000 -C
↪mps3_board. uart1_adapter_tx.ENABLE=true -C mps3_board. uart0_adapter_rx.ENABLE=true -C
↪mps3_board. uart0.rx_overrun_mode=0
```

2. After the FVP starts the following message will be shown in the FVP telnetterminal0:

```
Starting DMA350 Triggering example

-----
-----
Configure DMA350 for TX on UART1, then CPU goes to sleep.
Type in 10 character to this terminal
```

Select the FVP telnetterminal0 and type in 10 characters. The 10 characters are going to be echoed back in reverse order to the FVP telnetterminal1.

Copyright (c) 2022-2024, Arm Limited. All rights reserved.

FreeRTOS example to demonstrate the DMA-350 privileged and unprivileged APIs. The privileged task demonstrates a way of using of command linking feature. The unprivileged task demonstrates the usage of the unprivileged DMA API through a simple 2D example.

For detailed description of how privilege separation can be achieved with DMA-350, checkout [DMA-350 privilege separation](#)

Build steps

1. Build Secure TF-M with the following commands:

```
$ cmake -S <TF-M Source Dir> -B build/spe -DTFM_PLATFORM=arm/mps3/corstone310/fvp -DTFM_
↪PROFILE=profile_small
$ cmake --build build/spe -- -j$(nproc) install
```

2. Then to build the Non-Secure app:

```
$ cmake -S <path_to_this_example> -B build/nspe -DCONFIG_SPE_PATH=<absolute_path_to>/
↪build/spe/api_ns
$ cmake --build build/nspe -- -j$(nproc)
```

Copyright (c) 2022-2024, Arm Limited. All rights reserved.

TF-M Example Partition

The TF-M example partition is a simple Secure Partition implementation provided to aid development of new Secure Partitions.

It is an Application RoT, SFN model Secure Partition and implements an connection-based RoT Service.

Please refer to [PSA Firmware Framework 1.0](#) and [Firmware Framework for M 1.1 Extensions](#) for details of the attributes of Secure Partitions.

Please refer to [Adding Secure Partition](#) for more details of adding a new Secure Partition to TF-M.

File structure

```
.
├── CMakeLists.txt
├── README.rst
├── tfm_example_manifest_list.yaml
├── tfm_example_partition_api.c
├── tfm_example_partition_api.h
├── tfm_example_partition.c
└── tfm_example_partition.yaml
```

- `CMakeLists.txt`
The CMake file for building this example Secure Partitions. It is specific to the TF-M build system.
- `README.rst`
This document.
- `tfm_example_partition.yaml`
The manifest of this Secure Partition.
- `tfm_example_manifest_list.yaml`
The manifest list that describes the Secure Partition manifest of this Secure Partition. See [TF-M Manifest List](#) for details of manifest lists.
- `tfm_example_partition.c`
The core implementation of this Secure Partition.
- `tfm_example_partition_api.c`
The APIs for accessing the RoT Services provided by this Secure Partition.
- `tfm_example_partition_api.h`
The header file that declares the RoT Services APIs.

How to Build

It is recommended to build this example Secure Partition via out-of-tree build. It can minimize the changes to TF-M source code for building and testing the example.

To build, append the following extra build configurations to the CMake build commands.

- `-DTFM_PARTITION_EXAMPLE`

This is the configuration switch to enable or disable building this example. Set to `ON` to enable or `OFF` to disable.

- `-DTFM_EXTRA_PARTITION_PATHS`

Set it to the absolute path of this directory.

- `-DTFM_EXTRA_MANIFEST_LIST_FILES`

Set it to the absolute path of the manifest list mentioned above - `tfm_example_manifest_list.yaml`.

Refer to [Out-of-tree Secure Partition build](#) for more details.

Build steps for mps4/corstone315 platform

1. Build and install TF-M with the following command:

```
$ cmake -S <TF-M Source Dir> \
  -B build/spe_test \
  -DTFM_PLATFORM=arm/mps4/corstone315 \
  -DTFM_TOOLCHAIN_FILE=<TF-M Source Dir>/toolchain_<toolchain>.cmake \
  -DTFM_PARTITION_INTERNAL_TRUSTED_STORAGE=ON \
  -DTFM_PARTITION_CRYPTO=ON \
  -DTEST_NS=ON
  -DTEST_S=ON \
$ cmake --build build/spe_test -- -j$(nproc) install
```

2. Then build the example with the following:

```
$ cmake -S <this_example_path> \
  -B build/nspe_test \
  -DTFM_TOOLCHAIN=<toolchain> \
  -DCONFIG_SPE_PATH=$(shell pwd)/<build_dir>/spe_test/api_ns
$ cmake --build build/nspe_test -- -j$(nproc)
```

How to Test

To test the RoT Services, you need to build the APIs and call the service APIs somewhere.

If you want to add comprehensive tests using the TF-M test framework, please refer to [Adding TF-M Regression Test Suite](#).

Testing in NSPE

Any NSPE can be used to test the example RoT services. If you are using the `tf-m-tests` repo as NSPE, you can:

- Add the `tfm_example_partition_api.c` to `tfm_ns_api` CMake library.
- Add the current directory in the include directory of `tfm_ns_api`.
- Call the services APIs in the `test_app` function.

Testing in SPE

Testing in SPE is to test requesting the RoT Services in any Secure Partition.

- Add the example services to the `dependencies` attribute in the target Secure Partition's manifest.
- Call the services APIs somewhere in the Secure Partition, for example, in the entry function.

Note that the API source file has already been added in the `CMakeLists.txt`. There are no extra steps to build the APIs for testing in SPE.

References

[PSA Firmware Framework 1.0](#)

[Firmware Framework for M 1.1 Extensions](#)

[TF-M Manifest List](#)

[Out-of-tree Secure Partition build](#)

Copyright (c) 2020-2024, Arm Limited. All rights reserved.

Voice Activity Detection demo application

Trusted Firmware-M Voice Activity Detection Example Threat Model

Introduction

This document extends the generic threat model of Trusted Firmware-M (TF-M). This threat model provides an analysis of Voice Activity Detection (VAD) Example in TF-M and identifies general threats and mitigation.

Scope

TF-M supports diverse models and topologies. It also implements multiple isolation levels. Each case may focus on different target of evaluation (TOE) and identify different assets and threats. TF-M implementation consists of several secure services, defined as Root of Trust (RoT) service. Those RoT services belong to diverse RoT (Application RoT or PSA RoT) and access different assets and hardware. Therefore each RoT service may require a dedicated threat model.

This analysis only focuses on the assets and threats introduced by the VAD example. The TF-M implementation, topologies, or other RoT services are out of scope of this document.

Methodology

The threat modeling in this document follows the process listed below to build up the threat model.

- Target of Evaluation (TOE)
- Assets identification
- Data Flow Diagram (DFD)
- Threats prioritization
- Threats identification

TOE is the entity on which threat modeling is performed. The logic behind this process is to firstly investigate the TOE which could be a system, solution or use case. This first step helps to identify the assets to be protected in TOE.

According to TOE and assets, Trust Boundaries can be determined. The Data Flow Diagram (DFD) across Trust Boundaries is then defined to help identify the threats.

Those threats should be prioritized based on a specific group of principals and metrics. The principals and metrics should also be specified.

Target of Evaluation

A typical TF-M system diagram can be seen on [Generic Threat Model](#). TF-M is running in the Secure Processing Environment (SPE) and NS software is running in Non-secure Processing Environment (NSPE).

The TOE in this general model is the VAD Secure Partition and the interaction of peripherals, and NSPE. The VAD algorithm itself and its possible flaws are not in scope of this document, however the threats that such flaws can cause and its mitigations are in scope.

Asset identification

In this threat model, assets include the items listed below:

- Software RoT data, e.g.
 - Secure partition code and data
 - NSPE data stored in SPE
 - Data generated in SPE as requested by NSPE
 - Data flowing from peripherals to SPE
- Availability of entire RoT service
- Result of a RoT service

Data Flow Diagram

The list and details of data flows are described in the [Generic Threat Model](#). In addition to the data flows above, this use-case introduces a new data flow from a peripheral to the SPE. Although the peripheral resides within the SPE, the data from it is external so must be considered as data crossing a trust boundary. This Data flow will be labeled as DF7 from now on.

Note: All the other data flows across the Trusted Boundary besides the valid ones mentioned in the [Generic Threat Model](#) and above should be prohibited by default. Proper isolation must be configured to prevent NSPE directly accessing SPE.

Although the data flows are covered in general in the TF-M Generic Threat Model, for DF2-DF5, given the inner workings and flow of control in VAD partition, additional threats are also considered. Threats identified in the Generic Threat Model still applies.

Threat identification

Threat priority

Threat priority is indicated by the score calculated via Common Vulnerability Scoring System (CVSS) Version 3.1 [CVSS]. The higher the threat scores, the greater severity the threat is with and the higher the priority is.

CVSS scores can be mapped to qualitative severity ratings defined in CVSS 3.1 specification [CVSS_SPEC]. This threat model follows the same mapping between CVSS scores and threat priority rating.

This document focuses on *Base Score* which reflects the constant and general severity of a threat according to its intrinsic characteristics.

The *Impacted Component* defined in [CVSS_SPEC] refers to the assets listed in *Asset identification*.

Threats and mitigation list

This section lists generic threats and corresponding mitigation, based on the the analysis of data flows in *Data Flow Diagram*.

Threats are identified following STRIDE model. Please refer to [STRIDE] for more details.

The field CVSS *Score* reflects the threat priority defined in *Threat priority*. The field CVSS *Vector String* contains the textual representation of the CVSS metric values used to score the threat. Refer to [CVSS_SPEC] for more details of CVSS vector string.

Note: A generic threat may have different behaviors and therefore require different mitigation, in diverse TF-M models and usage scenarios.

This threat model document focuses on threats specific to the VAD partition. Similar threats might exist in the generic threat model with different consequence or severity. For the details of generic threats in general usage scenario, please refer to the [Generic Threat Model](#) document.

NSPE requests TF-M secure service

This section identifies threats on DF2 defined in *Data Flow Diagram*.

Table 1:: TFM-VAD-REQUEST-SERVICE-I-1

Index	TFM-VAD-REQUEST-SERVICE-I-1
Description	A malicious NS application may extract result of a VAD service request by measuring time while the service was unavailable for further request.
Justification	A malicious NS application may request VAD service to perform voice activity detection, while another legit NS app is doing so. By measuring how much time it takes for the service to become available, it can be extracted if there was voice activity or not.
Category	Information disclose
Mitigation	Not yet. Service could use non-blocking or callback based Implementation.
CVSS Score	2.9 (Low)
CVSS Vector String	CVSS:3.1/AV:L/AC:H/PR:N/UI:N/S:U/C:L/I:N/A:N

Table 2:: TFM-VAD-REQUEST-SERVICE-D-1

Index	TFM-VAD-REQUEST-SERVICE-D-1
Description	A Malicious NS applications may frequently call secure services to block secure service requests from other NS applications.
Justification	TF-M runs on IoT devices with constrained resource. Even though multiple outstanding NS PSA Client calls can be supported in system, the number of NS PSA client calls served by TF-M simultaneously are still limited. Therefore, if a malicious NS application or multiple malicious NS applications continue calling TF-M secure services frequently, it may block other NS applications to request secure service from TF-M. For VAD service request, this can have more consequence as the current implementation is blocking Secure thread.
Category	Denial of service
Mitigation	TF-M is unable to manage behavior of NS applications. Assets are not disclosed and TF-M is neither directly impacted in this threat. Repeatedly exploiting this vulnerability could disrupt and decrease the availability of TF-M and other secure services, but not completely. Because of this, the availability vector of the threat is considered high. It relies on NS OS to enhance scheduling policy and prevent a single NS application to occupy entire CPU time. It is beyond the scope of this threat model.
CVSS Score	6.2 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H

NS interrupts preempts SPE execution

This section identifies threats on DF5 defined in *Data Flow Diagram*.

Table 3:: TFM-VAD-NS-INTERRUPT-T-D-1

Index	TFM-VAD-NS-INTERRUPT-T-D-1
Description	An attacker may trigger spurious NS interrupts frequently to block SPE execution.
Justification	In single Armv8-M core scenario, an attacker may inject a malicious NS application or hijack a NS hardware to frequently trigger spurious NS interrupts to keep preempting SPE and block SPE to perform normal secure execution. Blocking VAD service long enough can cause loss of input data from peripherals to the service, possibly changing the return value of the service request.
Category	Tampering / Denial of service
Mitigation	It is out of scope of TF-M. Assets protected by TF-M won't be leaked. TF-M won't be directly impacted.
CVSS Score	5.1 (Medium)
CVSS Vector String	CVSS:3.1/AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:L/A:L

Data from peripherals to SPE

This section identifies threats on DF7 defined in *Data Flow Diagram*.

Table 4:: TFM-VAD-PERIPH-DATA-TO-SPE-T-D-1

Index	TFM-VAD-PERIPH-DATA-TO-SPE-T-D-1
Description	An attacker may gain ability to artificially modify the data and may trigger untested data paths within the voice activity detection algorithm.
Justification	TF-M is unable to prevent manipulation of external data, attacker might inject malicious data through the peripheral. The VAD algorithm is considered trusted, but given its complexity, might be subject to vulnerabilities within its data flow. By carefully crafted data, an attacker might be able to cause the failure of the VAD algorithm. It can also be used or gain in-depth knowledge of the algorithm, possibly making it prone to adversarial attacks. The attacker might also be able to read data accessible within the secure partition that the VAD algorithm is running in.
Category	Tampering / Denial of service
Mitigation	It is out of scope of TF-M to mitigate vulnerabilities within the VAD algorithm, however TF-M is responsible for properly isolating the algorithm within the secure partition, so vulnerabilities must not propagate.
CVSS Score	6.8 (Medium)
CVSS Vector String	CVSS:3.1/AV:P/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

Version control

Table 5:: Version control

Version	Description	TF-M version
v1.0	First version	TF-M v1.6.0

References

Copyright (c) 2020-2022 Arm Limited. All Rights Reserved.

A demo application for the AN552 FPGA showcasing:

- secure partition using MVE to speed up algorithms
- secure peripheral usage from secure partition with interrupt handling
- AWS cloud connectivity with OTA on the non-secure side

Brief Operation

After boot-up the application first checks whether Over-the-Air update (OTA) was initiated from AWS cloud. If yes, the OTA process is executed, otherwise the voice activity detection algorithm is started on the secure side. While the algorithm is running the non-secure side keep polling it for results. After a minute the algorithm is stopped, and the operation restarted with the OTA check again.

If the algorithm detects voice, a short audio sample (~100 ms) is recorded, and the highest energy frequency component is calculated. This frequency is written onto the serial line and it is sent to AWS cloud. Then the algorithm is restarted or the OTA check is started if the timeout is up.

By default the solution requires ethernet connectivity, it will not start the main operation until the network is up. This can be overwritten if the `-DVAD_AN552_NO_CONNECTIVITY=ON` cmake flag is defined. The effect is:

- No need for Ethernet connection.
- No need for IoT thing creation in AWS cloud and source update with its credentials.
- OTA check and AWS cloud communication is not executed.

HW requirements

- AN552 Version 2.0 FPGA image on MPS3 board.
- Ethernet connection with access to the internet. (Not needed if `-DVAD_AN552_NO_CONNECTIVITY=ON` is added for cmake.)
- 2 or 3 pole microphone connected into the audio connector. In case of a stereo microphone only the right channel is used.

Build instructions

AWS thing creation and source update with the credentials

By default it is required to create an IoT thing in AWS to run the application, but this can be skipped if `-DVAD_AN552_NO_CONNECTIVITY=ON` is added for `cmake`.

Create an IoT thing for your device

1. Login to your account and browse to the [AWS IoT console](#).
2. In the left navigation pane, choose **All devices**, and then choose **Things**.
3. Click on **Create things**.
4. Choose **Create single thing**.
5. At the **Specify thing properties** page add the name of your thing at **Thing name**. You will need to add the name later to your C code. Click **Next**.
6. At the **Configure device certificate** page choose **Auto-generate a new certificate**, and click **Next**.
7. The thing can be created by clicking on **Create thing** at the **Attach policies to certificate** page. The policy will be created at the next section.
8. Download the key files and the certificate, and make a note of the name of the certificate.
9. Activate your certificate if it is not active by default.

Create a policy

For the sake of simplicity in this example a very permissive Policy is created, for production usage a more restrictive one is recommended.

1. In the navigation pane of the AWS IoT console, choose **Security**, and then choose **Policies**.
2. At the **Policies** page, choose **Create policy**.
3. At the **Create a policy** page, enter a name for the policy.
4. At the **Policy document** click on **JSON**, and paste the following snippet into the **Policy document** textbox, then click on **Create**. (**Region** and **Account ID** must be updated.)

```
{
"Version": "2012-10-17",
"Statement": [
  {
    "Effect": "Allow",
    "Action": [
      "iot:Connect",
      "iot:Publish",
      "iot:Subscribe",
      "iot:Receive"
    ],
    "Resource": "arn:aws:iot:<Region>:<Account ID without dashes>:*"
  }
]
```

(continues on next page)

(continued from previous page)

```
]
}
```

Attach the created policy to your thing

1. In the left navigation pane of the AWS IoT console, choose **Secure**, and then choose **Certificates**. You should see the certificate that you have created earlier.
2. Click on the three dots next to the certificate and choose **Attach policy**.
3. In the **Attach policies to certificate(s)** window choose the created policy and click **Attach**.

Update source with thing credentials

Edit `examples/vad_an552/ns_side/amazon-freertos/aws_clientcredential.h` file and set the value of the following macros:

- `clientcredentialMQTT_BROKER_ENDPOINT`, set this to the endpoint name of your amazon account. To find this go to the AWS IoT console page and in the left navigation pane click on **Settings**. The Endpoint can be found under **Device data endpoint**.
- `clientcredentialIOT_THING_NAME`, set this to the name of the created thing.

Recreate or update `examples/vad_an552/ns_side/amazon-freertos/aws_clientcredential_keys.h` with the downloaded certificate and keys.

Recreate with the html tool from Amazon-FreeRTOS:

1. Clone [Amazon-FreeRTOS](#).
2. Open `Amazon-FreeRTOS/tools/certificate_configuration/CertificateConfigurator.html` in your browser.
3. Upload the downloaded certificate and the private key.
4. Click on **Generate** and save `aws_clientcredential_keys.h`
5. Download the file and update `examples/vad_an552/ns_side/amazon-freertos/aws_clientcredential_keys.h` with it.

Alternatively, the file can be updated by hand by setting the values of the following macros:

- `keyCLIENT_CERTIFICATE_PEM`, content of `<your-thing-certificate-unique-string>-certificate.pem.crt`.
- `keyCLIENT_PRIVATE_KEY_PEM`, content of `<your-thing-certificate-unique-string>-private.pem.key`.
- `keyCLIENT_PUBLIC_KEY_PEM`, content of `<your-thing-certificate-unique-string>-public.pem.key`.

Running TF-M build

For building TF-M's build system is used with the following mandatory CMAKE flags:

```
-DTFM_PLATFORM=arm/mps3/corstone300/an552
-DNS_EVALUATION_APP_PATH=<path-to-tf-m-extras-repo>/examples/vad_an552/ns_side
-DTFM_EXTRA_PARTITION_PATHS=<path-to-tf-m-extras-repo>/partitions/vad_an552_sp/
-DTFM_EXTRA_MANIFEST_LIST_FILES=<path-to-tf-m-extras-repo>/partitions/vad_an552_sp/extra_
↪manifest_list.yaml
-DPROJECT_CONFIG_HEADER_FILE=<path-to-tf-m-extras-repo>/examples/vad_an552/ns_side/
↪project_config.h
-DTFM_PARTITION_FIRMWARE_UPDATE=ON -DMCUBOOT_DATA_SHARING=ON
-DMCUBOOT_UPGRADE_STRATEGY=SWAP_USING_SCRATCH
-DMCUBOOT_IMAGE_NUMBER=1 -DMCUBOOT_SIGNATURE_KEY_LEN=2048
-DCONFIG_TFM_ENABLE_MVE=ON -DCONFIG_TFM_SPM_BACKEND=IPC
-DPLATFORM_HAS_FIRMWARE_UPDATE_SUPPORT=ON -DTFM_PARTITION_PLATFORM=ON
-DTFM_PARTITION_CRYPTO=ON -DTFM_PARTITION_INTERNAL_TRUSTED_STORAGE=ON
-DTFM_PARTITION_PROTECTED_STORAGE=ON -DMCUBOOT_CONFIRM_IMAGE=ON
```

The application also can be run without MVE support, in that case the `-DCONFIG_TFM_ENABLE_MVE=ON` flags should be omitted, and the `configENABLE_MVE` can be set to `0` in the `ns_side/amazon-freertos/FreeRTOSConfig.h` file. Our measurements showed that MVE speeds up the frequency calculation by 10 times with release GCC build.

You can check TF-M's build instructions [here](#).

Running the application

It is covered by the generic TF-M run instructions for AN552 [here](#).

Testing the voice algorithm

Start up the board, wait until `==== Start listening ====` is written on the serial console and start talking, or make some noise. You can check that the `Voice detected with most energy at X Hz` message is written onto the serial console, and the same message is sent to AWS cloud.

For checking the AWS messages:

1. In the left navigation pane of the AWS IoT console, choose `Test`.
2. Define `<Name of your thing>/vad_an552` as the topic filter.
3. Click on `Subscribe`.
4. Once a message is sent to AWS cloud you should see it on this page.

Note: For this test it is recommended to find a quiet environment, because any noise can trigger the voice activity algorithm.

For testing the frequency calculation pure sine signals should be used, the accuracy is about ± 100 Hz.

Testing Amazon AWS OTA

To run an OTA update a new image must be created with higher version number. This can be easily done by rebuilding the solution with the following cmake flag: `-DMCUBOOT_IMAGE_VERSION_S=2.1.0`. (The version itself can be anything, but must be higher than the version of the currently running image.) The `-DMCUBOOT_CONFIRM_IMAGE` flag should be set to OFF in the new image build config, because the demo going to confirm the new image after downloading it.

The image signature must be extracted from the final binary, can be done by openssl running the following commands in the build directory:

1. `openssl dgst -sha256 -binary -out update-digest.bin tfm_s_ns_signed.bin`
2. `openssl pkeyutl -sign -pkeyopt digest:sha256 -pkeyopt rsa_padding_mode:pss -pkeyopt rsa_mgf1_md:sha256 -inkey <path to tfm source>/bl2/ext/mcuboot/root-RSA-2048.pem -in update-digest.bin -out update-signature.bin`
3. `openssl base64 -A -in update-signature.bin -out update-signature.txt`

Once the signature extracted into `update-signature.txt` file, the OTA job can be created:

1. Create an Amazon S3 bucket to store your update.
2. Create an OTA Update service role.
3. Create an OTA user policy.
4. Go to AWS IoT web interface and choose Manage and then Jobs.
5. Click the create job button and select Create FreeRTOS OTA update job.
6. Give it a name and click next.
7. Select the device to update (the Thing you created in earlier steps).
8. Select MQTT transport only.
9. Select Use my custom signed file.
10. Paste the signature string from the `update-signature.txt` file. Make sure that it is pasted as it is without any whitespace characters.
11. Select SHA-256 and RSA algorithms.
12. For Path name of code signing certificate on device put in `0` (the path is not used).
13. Select upload new file and select the signed update binary `tfm_s_ns_signed.bin`.
14. Select the S3 bucket you created to upload the binary to.
15. For Path name of file on device put in `combined image`.
16. As the role, select the OTA role you created.
17. Click next.
18. Click next, your update job is ready and running. If your board is running (or the next time it will be turned on) the update will be performed.

After the update happened the system resets, and the image version is written onto the serial console. That way the update can be verified.

Note: The OTA process only updates the image stored in RAM, so if the MPS3 board is power cycled the system will boot up with the original image. The FPGA at power-on loads the application image from the SD card to RAM, and

the SD card content is not changed during OTA.

Copyright (c) 2021-2023, Arm Limited. All rights reserved.

TF-M Example Application

This **tf-m-example-ns-app** directory provides a bare metal example NS application, demonstrating how to use the artifacts exported by TF-M build.

The application outputs “Hello TF-M world” and uses a PSA service function to demonstrate that a NS application can be successfully run.

The Build Process

1. Clone the TF-M repository at anywhere, assume the root directory is <TF-M Source Dir>
2. Build the TF-M with the following command:

```
cmake -S <TF-M Source Dir> -B build/spe -DTFM_PLATFORM=arm/mps2/an521 -DTFM_
↪PROFILE=profile_small
cmake --build build/spe -- install
```

3. The files necessary to build TF-M will appear in `build/spe/api_ns`. The most important elements are:
 - Bootloader: `/bin/bl2.*`
 - Secure (S) side binary image: `/bin/tfm_s.*`
 - PSA API for Non-Secure application: `/interface/*`
 - Files for building the Non-Secure application with different toolchains (see [Toolchains](#)): `/cmake/toolchain_ns_*.cmake`
4. Build this example TF-M application:

```
cmake -S <path_to_this_example> -B build/nspe -DCONFIG_SPE_PATH=<absolute_path_to>/build/
↪spe/api_ns
cmake --build build/nspe
```

5. In output you will get:
 - Non-secure (NS) application in `build/nspe/bin/tfm_ns.*`
 - Combined S + NS binary `build/nspe/bin/tfm_s_ns.bin`
 - Combined S + NS and signed binary `build/nspe/bin/tfm_s_ns.bin`

Run the Application

The application binary shall be loaded and launched on address `0x10080000`. The application can be run using the SSE-200 fast-model using FVP_MPS2_AEMv8M provided by Arm Development Studio. Add `b12.axf` and `tfm_s_ns_signed.bin` to the symbol files in the Debug Configuration menu. The following output you shall find in a serial terminal:

```
Non-Secure system starting...
Hello TF-M world
PSA Framework Version = 1.1
Testing psa get random number...
1: psa_generate_random() = 254
2: psa_generate_random() = 214
3: psa_generate_random() = 129
4: psa_generate_random() = 226
5: psa_generate_random() = 102
End of TF-M example App
```

Copyright (c) 2023, Arm Limited. All rights reserved.

The list and simple introduction of the examples in this folder.

DMA-350 for FreeRTOS

- DMA-350 Secure tests
- Non-secure DMA-350 examples for the Corstone-310 FVP platform

Maintainers

- Bence Balogh <bence.balogh@arm.com>
- Mark Horvath <mark.horvath@arm.com>

Example Partition

A simple secure partition implementation.

Maintainers

Jianliang Shen <jianliang.shen@arm.com>

Voice Activity Detection

Example application for the AN552 FPGA image, details can be found [here](examples/vad_an552/readme.rst).

Maintainers

- Gabor Toth <gabor.toth@arm.com>
- Mark Horvath <mark.horvath@arm.com>

Copyright (c) 2021-2022, Arm Limited. All rights reserved.

Copyright (c) 2021-2024, Arm Limited. All rights reserved.

BIBLIOGRAPHY

[CVSS] Common Vulnerability Scoring System Version 3.1 Calculator

[CVSS_SPEC] CVSS v3.1 Specification Document

[STRIDE] The STRIDE Threat Model