User Guide

Release v2.1.0-RC2

ARM CE-OSS

May 03, 2024

OVERVIEW

1 CMSIS TF-M Packs **1**
- 1.1 Prerequisites 1
- 1.2 Create packs 1

2 Code size analyze tool **3**
- 2.1 Install dependency 4
- 2.2 Usage 4
- 2.3 Code size diff tool 7

3 FIH TEST TOOL **9**
- 3.1 Description 9
- 3.2 Dependencies 10
- 3.3 Usage of the tool 10
- 3.4 Fault types 10
- 3.5 Working with results 11

4 Initial Attestation Verifier **13**
- 4.1 Installation 13
- 4.2 Usage 13
- 4.3 Mac0Message 16
- 4.4 Testing 16
- 4.5 Development Scripts 17
- 4.6 Adding new token type 17

5 IRQ test tool **19**
- 5.1 Introduction 19
- 5.2 How the tool works 19
- 5.3 Input files 21
- 5.4 How to run the example 23
- 5.5 Implementation details 25
- 5.6 Possible further improvements 26

6 Library dependency trace tool **27**
- 6.1 Install graphviz 27
- 6.2 Usage 28

7 Profiler tool and TF-M Profiling **33**
- 7.1 TF-M Profiling Build Instructions 33
- 7.2 Profiler Integration Reference 34
- 7.3 TF-M Profiling Cases 36

8 SQUAD metrics dashboard **39**
8.1 SQUAD Presentation 39
8.2 Metrics and configurations 39
8.3 CI integration 40
8.4 Adding a new platform 40

9 Static Checking Framework **41**
9.1 Instructions 41

10 TF_Fuzz **47**

11 Using docker to build TF-M **51**
11.1 Configuration Parameters 51
11.2 Building an image 51
11.3 Running an image 51

CHAPTER

ONE

CMSIS TF-M PACKS

This repository contains tools and data to create TF-M CMSIS-Packs.

1.1 Prerequisites

- bash compatible shell (under Windows, use for example git bash)
- Git
- 7-Zip
- Python v3.6 or later with pip package manager
- Doxygen v1.8.0 or later (for building documentation)
- Graphviz v2.38.0 or later (for building documentation)
- PlantUML v1.2018.11 or later in PLANTUML_JAR_PATH (for building documentation)
- Java runtime environment 1.8 or later (for running PlantUML)
- CMSIS Pack installed in CMSIS_PACK_ROOT (for PackChk utility)
- xmllint in path (XML schema validation; available only for Linux)

1.2 Create packs

1. Open a bash compatible shell
2. Run `./setup.sh` script. The script will:
    - install the required python packages
    - clone the trusted-firmware-m repository
    - clone the tf-m-tests repository
    - clone the mcuboot repository
    - merge mcuboot into trusted-firmware-m
    - apply patches for trusted-firmware-m
    - apply patches for tf-m-tests
    - generate template based files
    - setup tf-m-tests (copy/move files from trusted-firmware-m)

    - merge addon files for trusted-firmware-m
    - merge addon files for tf-m-tests

3. Generate TF-M documentation:
    - setup path variable for PlantUML: `export PLANTUML_JAR_PATH=<plantuml_Path>/plantuml.jar`
    - run `gen_doc.sh` script
4. Generate CMSIS-Packs:
    - TFM:
        - go to `./trusted-firmware-m` directory
        - run `gen_pack.sh` script
        - generated pack is available in the `output` directory
    - TFM-Test:
        - go to `./tf-m-tests` directory
        - run `gen_pack.sh` script
        - generated pack is available in the `output` directory
    - V2M-MPS2_SSE_200_TFM-PF (TF-M Platform support for MPS2):
        - run `setup_mps2.sh` script
        - go to `./tf-m-platform-mps2` directory
        - run `gen_pack.sh` script
        - generated pack is available in the `output` directory
        - note: this pack should not be published and used only for testing TF-M
5. Run `./clean.sh` script to delete all intermediate and generated files

---

CHAPTER

TWO

CODE SIZE ANALYZE TOOL

Table of Contents

- *Code size analyze tool*
  - *Install dependency*
    - *Linux*
    - *Windows*
  - *Usage*
    - *Create database*
    - *UI mode*
    - *Terminal mode*
      - *Dump detail information*
      - *Search specific function or data*
      - *Sort Table*
  - *Code size diff tool*

These are scripts to dump ARMCLANG complie map file detail information and get the change between two different build results. The features are:

- Get a database to implement data tables from ARMCLANG or GNUARM map file.
- Supply an UI for developers to scan sorted information of sections, libraries, object files, functions and data. It can help analyze the biggest or smallest targets in certain image. It can also help search functions or data to locate their address and detail information.
- Diff two databases to see the code size increasement or decreasement.

2.1 Install dependency

2.1.1 Linux

Install curse and database.

```
sudo apt-get install python3-dev libmysqlclient-dev libncursesw6 libncurses6 libsqlite3-
↪dev
pip install mysqlclient XlsxWriter
```

2.1.2 Windows

In power shell, use pip to install packages.

```
pip install mysqlclient XlsxWriter windows-curses
```

2.2 Usage

The commands of code size analyze tool usage are:

```
usage: code_size_analyze.py [-h] [--gnuarm | --armcc] [--db-name <data.db>]
                            [-u | -S | -s | -l | -o | -f | -d |
                             --dump-sec <sec> | --dump-lib <lib> |
                             --dump-obj <obj> | --dump-func <func> |
                             --dump-data <data> | --search-func <func> |
                             --search-data <data>]
                            [--sort-by-size | --sort-by-name |
                             --sort-by-obj | --sort-by-lib |
                             --sort-by-sec] [--desc | --asc]
                             file_input

positional arguments:
    file_input              map or database file

optional arguments:
    -h, --help              show this help message and exit
    --gnuarm                GNUARM model
    --armcc                 ARMCLANG model
    --db-name <data.db>     database name to save
    -u, --ui                start UI to analyze
    -S, --Summary           show summary message
    -s, --list-section      list section
    -l, --list-library      list library
    -o, --list-obj          list object file
    -f, --list-function     list function
    -d, --list-data         list data
    --dump-sec <sec>        dump section
    --dump-lib <lib>        dump library
    --dump-obj <obj>        dump object file
```

(continues on next page)

(continued from previous page)

```
    --dump-func <func>    dump function
    --dump-data <data>    dump data
    --search-func <func>  search function
    --search-data <data>  search data
    --sort-by-size        list by size order
    --sort-by-name        list by name order
    --sort-by-obj         list by object file name order
    --sort-by-lib         list by library file name order
    --sort-by-sec         list by section name order
    --desc                sort with desc order
    --asc                 sort with asc order
```

2.2.1 Create database

It is required to input map file path for the script before show the UI. One of the options like `--gnuarm` or `--armcc` is required.

The default database name created is `data.db`. Use `--db-name` to name the output file if necessary. For example, saving two different databases to compare later.

```
$: python code_size_analyze.py tfm_s.map <--gnuarm|--armcc> --db-name tfm_s.db
```

2.2.2 UI mode

The script ui.py supplies a menu to choose what developers may be interested. You can enter UI mode by analyzing map file directly or by importing database file path. The latter way is suggested as it runs more quickly.

```
$: python code_size_analyze.py tfm_s.map <--gnuarm|--armcc> -u
$: python code_size_analyze.py tfm_s.db -u
```

There are several keys to use UI.

- UP: Move UP, mouse scrolling up is same.
- DOWN: Move down, mouse scrolling down is same.
- RIGHT: Move right.
- LEFT: Move left.
- Enter: Move to next page if it can be unfolded.
- `Q` or `q`: Escape to previous page or close script if it in top menu.
- `s` or S: Enter output file name to save the content of current page.
- `:` : Start search and enter the function or data name.

2.2.3 Terminal mode

In terminal mode, it is better to analyze database file rather than map file.

Dump detail information

You can get the list of all sections, libraries, object files, functions or data. You can also dump the specific symbol with the name.

```
$: python code_size_analyze.py tfm_s.map --armcc --db-name test.db -S
───────────────────────────────────────────────
Code size       : 56676         55.35   KB
-----------------------------------------------
RO data         : 3732          3.64    KB
RW data         : 204           0.20    KB
ZI data         : 24588         24.01   KB
Flash size      : 60612         59.19   KB = Code + RO + RW
RAM size        : 24792         24.21   KB = RW + ZI
───────────────────────────────────────────────

$: python code_size_analyze.py tfm_s.db -s
$: python code_size_analyze.py tfm_s.db --dump-sec <sec>
```

Search specific function or data

You can search the target with keyword in command line. For example:

```
$: python code_size_analyze.py tfm_s.db --search-func <func>
$: python code_size_analyze.py tfm_s.db --search-data <data>
```

Sort Table

You can sort the messages in terminal mode. The script supplies five options and two orders. For example:

```
$: python code_size_analyze.py tfm_s.db -l --sort-by-size --asc
────────────────────────────────────────────────────────────────────────────────────────────────────
Name                                              Flash size  RAM size    Code        RO␣
↪data      RW data     ZI data     Inc. data   Debug
-----------------------------------------------------------------------------------------
↪-----------------------------------------------------------
libtfm_qcbor_s.a                                  758         0           758         0 ␣
↪          0           0           4           17046
libtfm_sprt.a                                     1016        0           1016        0 ␣
↪          0           0           0           41004
c_w.l                                             1248        96          1248        0 ␣
↪          0           96          86          1892
libtfm_psa_rot_partition_attestation.a            2497        557         2492        5 ␣
↪          0           557         68          51865
libtfm_spm.a                                      4112        657         3932       ␣
↪136         44          613         168         52958
libtfm_psa_rot_partition_its.a                    5090        116         5030        32␣
↪          28          88          28          49804
```

(continues on next page)

(continued from previous page)

```
libtfm_psa_rot_partition_crypto.a                   6062        3232        6062        0 ␣
↪          0           3232        36          92472
libplatform_s.a                                     6486        316         5582          ␣
↪780         124         192         404         94887
libmbedcrypto.a                                     28408       2292        26138         ␣
↪2262        8           2284        1066        226489
```

Not all symbols support to be sorted by the whole five options, refer to the table to get more information.

Options	Section	Library	Object files	Function	Data
–sort-by-size					
–sort-by-name					
–sort-by-sec					
–sort-by-lib					
–sort-by-obj					

2.3 Code size diff tool

Use `code_size_diff.py` to diff two diffrent build results with same compiler.

```
usage: code_size_diff.py [-h] (-S | -f | -d | -o | -l) based_db compared_db

positional arguments:
    based_db             based databse
    compared_db          compared databse

optional arguments:
    -h, --help            show this help message and exit
    -S, --diff-Summary    diff summary
    -f, --diff-function   diff function
    -d, --diff-data       diff data
    -o, --diff-obj        diff object file
    -l, --diff-lib        diff library
```

Firstly, use `code_size_analyze.py` to prepare two different databases. Then compare two database with the diff tool, the branch1 is base.

```
$: python code_size_diff.py output/branch1.db output/branch2.db -S
Code size:  +++         48928   B               47.78   KB
RO data:    +++         29440   B               28.75   KB
RW data:    ---         64      B               0.06    KB
ZI data:    ---         500     B               0.49    KB
Flash size: +++         78304   B               76.47   KB
RAM size:   ---         564     B               0.55    KB
```

CHAPTER

THREE

FIH TEST TOOL

This directory contains a tool for testing the fault injection mitigations implemented by TF-M.

3.1 Description

The tool relies on QEMU (simulating the AN521 target), and GDB.

The tool will:

- first compile a version of TF-M with the given parameters.
- Then setup some infrastructure used for control of a QEMU session, including a Virtual Hard Drive (VHD) to save test state and some FIFOs for control.
- Then run GDB, which then creates a QEMU instance and attaches to it via the GDBserver interface and the FIFOs.
- Run through an example execution of TF-M, conducting fault tests.
- Output a JSON file with details of which tests where run, and what failed.

The workflow for actually running the test execution is as follows:

- Perform setup, including starting QEMU.
- **Until the end_breakpoint is hit:**
  - Execute through the program until a *test_location* breakpoint is hit.
  - Save the execution state to the QEMU VHD.
  - Enable all the *critical point breakpoints*.
  - **Run through until the end_breakpoint to save the "known good" state. The**
    state saving is described below.
  - **For each of the fault tests specified:**
    - Load the test_start state, so that a clean environment is present.
    - Perform the fault.
    - **Run through, evaluating any *critical memory* at every**
      critical point breakpoint and saving this to the test state.
    - **Detect any failure loop states, QEMU crashes, or the end breakpoint,**
      and end the test.
    - Compare the execution state of the test with the "known good" state.

- Load the state at the start of the test.
- Disable all the *critical point breakpoints*.

The output file will be inside the created TFM build directory. It is named *results.json* The name of the created TFM build dir will be determined by the build options. For example *build_GNUARM_debug_OFF_2/results.json*

3.2 Dependencies

- qemu-system-arm
- gdb-multiarch (with python3 support)
- python3.7+
- python packages detailed in requirements.txt

The version of python packaged with gdb-multiarch can differ from the version of python that is shipped by the system. The version of python used by gdb-multiarch can can be tested by running the command: *gdb-multiarch -batch -ex "python import sys; print(sys.version)"*. If this version is not greater than or equal to 3.7, then gdb-multiarch may need to be upgraded. Under some distributions, this might require upgrading to a later version of the distribution.

3.3 Usage of the tool

Options can be determined by using

`./fih_test --help`

In general, executing *fih_test* from a directory inside the TF-M source directory (*<TFM_DIR>/build*), will automatically set the SOURCE_DIR and BUILD_DIR variables / arguments correctly.

For example:

```
cd <TFM_DIR>
mkdir build
cd build
<Path to>/fih_test -p LOW

# Test with certain function
<Path to>/fih_test -p LOW -l 2 -f "tfm_hal_set_up_static_boundaries"

# Build the AXF file again if the source code has been changed
<Path to>/fih_test -p LOW -l 2 -r
```

3.4 Fault types

The types of faults simulated is controlled by `faults/__init__.py`. This file should be altered to change the fault simulation parameters. To add new fault types, new fault files should be added to the `faults` directory.

Currently, the implemented fault types are:

- Setting a single register (from r0 to r15) to a random uint32
- Setting a single register (from r0 to r15) to zero

- Skipping between 1 and 7 instructions

All of these will be run at every evaluation point, currently 40 faults per evaluation point.

3.5 Working with results

Results are written as a JSON file, and can be large. As such, it can be useful to employ dedicated tools to parse the JSON.

The use of *jq <https://stedolan.github.io/jq/>* is highly recommended. Full documentation of this program is out of scope of this document, but instructions and reference material can be found at the linked webpage.

For example, to find the amount of passes:

```
cat results.json | jq 'select(.passed==true) | .passed' | wc -l
```

And the amount of fails:

```
cat results.json | jq 'select(.passed==false) | .passed' | wc -l
```

To find all the faults that caused failures, and the information about where they occurred:

```
cat results.json | jq 'select(.passed==false) | {pc:  .pc, file:  .file, line:  .line,
asm:  .asm, fault:  .fault}'
```

CHAPTER

FOUR

INITIAL ATTESTATION VERIFIER

This is a set of utility scripts for working with PSA Initial Attestation Token, the structure of which is described here:

https://tools.ietf.org/html/draft-tschofenig-rats-psa-token-05

The following utilities are provided:

check_iat
Verifies the structure, and optionally the signature, of a token.

compile_token
Creates a (optionally, signed) token from a YAML descriptions of the claims.

decompile_token
Generates a YAML descriptions of the claims contained within a token. (Note: this description can then be compiled back into a token using compile_token.)

4.1 Installation

You can install the script using pip:

```
# Inside the directory containg this README
pip3 install .
```

This should automatically install all the required dependencies. Please see `setup.py` for the list of said dependencies.

4.2 Usage

Note: You can use `-h` flag with any of the scripts to see their usage help.

4.2.1 check_iat

After installing, you should have `check_iat` script in your `PATH`. The script expects two parameters:

- a path to the signed IAT in COSE format
- the token type

You can find an example in the `tests/data` directory.

The script will extract the COSE payload and make sure that it is a valid IAT (i.e. all mandatory fields are present, and all known fields have correct size/type):

```
$ check_iat -t PSA-IoT-Profile1-token tests/data/iat.cbor
Token format OK
```

If you want the script to verify the signature, you need to specify the file containing the signing key in PEM format using -k option. The key used to sign tests/data/iat.cbor is inside tests/data/key.pem.

```
$ check_iat -t PSA-IoT-Profile1-token -k tests/data/key.pem tests/data/iat.cbor
Signature OK
Token format OK
```

You can add a -p flag to the invocation in order to have the script print out the decoded IAT in JSON format. It should look something like this:

```
{
    "INSTANCE_ID": "b'0107060504030201000F0E0D0C0B0A0908171615141312111101F1E1D1C1B1A1918'
↪",
    "IMPLEMENTATION_ID": "b
↪'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'",
    "CHALLENGE": "b'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'",
    "CLIENT_ID": 2,
    "SECURITY_LIFECYCLE": "SL_SECURED",
    "PROFILE_ID": "http://example.com",
    "BOOT_SEED": "b'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'",
    "SW_COMPONENTS": [
        {
            "SW_COMPONENT_TYPE": "BL",
            "SIGNER_ID": "b
↪'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'",
            "SW_COMPONENT_VERSION": "3.4.2",
            "MEASUREMENT_VALUE": "b
↪'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'",
            "MEASUREMENT_DESCRIPTION": "TF-M_SHA256MemPreXIP"
        },
        {
            "SW_COMPONENT_TYPE": "M1",
            "SIGNER_ID": "b
↪'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'",
            "SW_COMPONENT_VERSION": "1.2",
            "MEASUREMENT_VALUE": "b
↪'07060504030201000F0E0D0C0B0A090817161514131211101F1E1D1C1B1A1918'"
        },
        {
            "SW_COMPONENT_TYPE": "M2",
```

(continues on next page)

(continued from previous page)

```
            "SIGNER_ID": "b
↪'0706050403020100OF0E0D0C0B0A0908171615141312111O1F1E1D1C1B1A1918'",
            "SW_COMPONENT_VERSION": "1.2.3",
            "MEASUREMENT_VALUE": "b
↪'07060504030201000F0E0D0C0B0A09081716151413121110 1F1E1D1C1B1A1918'"
        },
        {
            "SW_COMPONENT_TYPE": "M3",
            "SIGNER_ID": "b
↪'07060504030201000F0E0D0C0B0A0908171615141312111O1F1E1D1C1B1A1918'",
            "SW_COMPONENT_VERSION": "1",
            "MEASUREMENT_VALUE": "b
↪'07060504030201000F0E0D0C0B0A0908171615141312111O1F1E1D1C1B1A1918'"
        }
    ]
}
```

4.2.2 compile_token

You can use this script to compile a YAML claims description into a COSE-wrapped CBOR token:

```
$ compile_token -t PSA-IoT-Profile1-token -k tests/data/key.pem tests/data/iat.yaml >␣
↪sample_token.cbor
```

No validation is performed as part of this, so there is no guarantee that a valid IAT will be produced.

You can omit the -k option, in which case, the resulting token will not be signed, however it will still be wrapped in COSE "envelope". If you would like to produce a pure CBOR encoding of the claims without a COSE wrapper, you can use -r flag.

4.2.3 decompile_token

Decompile an IAT (or any COSE-wrapped CBOR object – *no validation* is performed as part of this) into a YAML description of its claims.

```
$ decompile_token -t PSA-IoT-Profile1-token tests/data/iat.cbor
boot_seed: !!binary |
  BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
challenge: !!binary |
  BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
client_id: 2
implementation_id: !!binary |
  BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
instance_id: !!binary |
  AQcGBQQDAgEADw4NDAsKCQgXFhUUExIREB8eHRwbGhkY
profile_id: http://example.com
security_lifecycle: SL_SECURED
sw_components:
- measurement_description: TF-M_SHA256MemPreXIP
  measurement_value: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
```

(continues on next page)

(continued from previous page)

```
  signer_id: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  sw_component_type: BL
  sw_component_version: 3.4.2
- measurement_value: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  signer_id: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  sw_component_type: M1
  sw_component_version: '1.2'
- measurement_value: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  signer_id: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  sw_component_type: M2
  sw_component_version: 1.2.3
- measurement_value: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  signer_id: !!binary |
    BwYFBAMCAQAPDg0MCwoJCBcWFRQTEhEQHx4dHBsaGRg=
  sw_component_type: M3
  sw_component_version: '1'
```

This description can then be compiled back into CBOR using `compile_token`.

4.3 Mac0Message

By default, the expectation is that the message will be wrapped using Sign1Message COSE structure, however, the alternative Mac0Message structure that uses HMAC with SHA256 algorithm rather than a signature is supported via the `-m mac` flag:

```
$ check_iat -t PSA-IoT-Profile1-token -m mac -k tests/data/hmac.key tests/data/iat-hmac.
↪cbor
Signature OK
Token format OK
```

4.4 Testing

Tests can be run using `nose2`:

```
pip install nose2
```

Then run by executing `nose2` in the root directory.

4.5 Development Scripts

The following utility scripts are contained within `dev_scripts` subdirectory and were utilized in development of this tool. They are not need to use the iat-verifier script, and can generally be ignored.

```
./dev_scripts/generate-key.py OUTFILE
```

Generate an ECDSA (NIST256p curve) signing key and write it in PEM format to the specified file.

```
./dev_scripts/generate-sample-iat.py KEYFILE OUTFILE
```

Generate a sample token, signing it with the specified key, and writing the output to the specified file.

Note: This script is deprecated – use `compile_token` (see above) instead.

4.6 Adding new token type

1. Create a file with the claims for the new token type in *tf-m-tools/iat-verifier/iatverifier*.
    - For each claim a new class must be created that inherits from `AttestationClaim` or from one of its descendants
    - `CompositeAttestClaim` is descendants of `AttestationClaim`, for details on how to use it see the documentation in the class definition.
    - For each claim, the methods `get_claim_key(self=None)`, `get_claim_name(self=None)` must be implemented.
    - Other methods of `AttestationClaim` are optional to override.
    - Any claim that inherits from `AttestationClaim` might have a `verify` method (`def verify(self, token_item):`). This method is called when the `verify()` method of a `TokenItem` object is called. `TokenItem`'s `verify()` method walks up the inheritance tree of the `claim_type` object's class in that `TokenItem`. If a class in the walk path has a `verify()` method, calls it. For further details see `TokenItem`'s `_call_verify_with_parents()` method.

      Any verify method needs to call `AttestationClaim`'s `error()` or `warning()` in case of a problem. If the actual class inherits from `AttestationTokenVerifier` this can be done like `self.error('Meaningful error message.')`. In other cases `self.verifier.error('Meaningful error message.')`
2. Create a file for the new token in *tf-m-tools/iat-verifier/iatverifier*.
    - Create a new class for the token type. It must inherit from the class `AttestationTokenVerifier`.
    - Implement `get_claim_key(self=None)` and `get_claim_name(self=None)`
    - Implement the `__init__(self, ...)` function. This function must create a list with the claims that are accepted by this token. (Note that the `AttestationTokenVerifier` class inherits from `AttestationClaim`. this makes it possible to create nested token). Each item is the list is a tuple:
        - first element is the class of the claim
        - Second is a dictionary containing the `__init__` function parameters for the claim
            * the key is the name of the parameter
            * the value is the value of the parameter

The list of claims must be passed to the init function of the base class.

For example see *iat-verifier/iatverifier/cca_token_verifier.py*.

3. Add handling of the new token type to the `check_iat`, `decompile_token`, and `compile_token` scripts.

CHAPTER

FIVE

IRQ TEST TOOL

5.1 Introduction

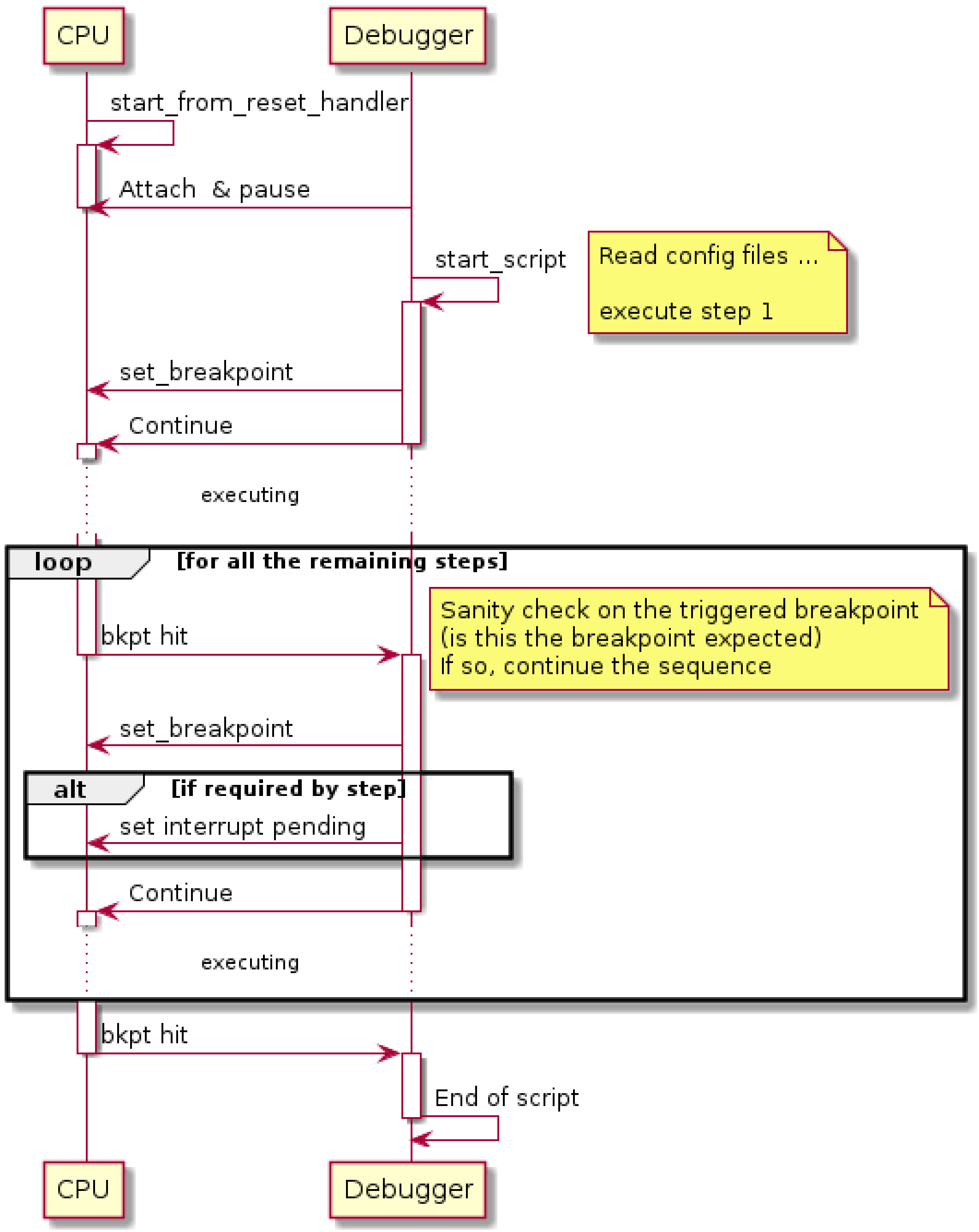
This tool is to test interrupt handling in TF-M. Testing different interrupt scenarios is important as the ARMv8-M architecture does complex operations when interrupt happens, especially when security boundary crossing happens. These operations need to be considered by the TF-M implementation, as in a typical use case there is a separate scheduler on the Non-Secure and the secure side as well, and the SPM needs to maintain a consistent state, which might not be trivial.

The aim of the tool is to be able to test scenarios, that are identified to be problematic, in a reproducible way, and do this in an automated way, so regular regression testing can have a low cost.

5.2 How the tool works

The tool is a set of Python scripts which need to be run **inside** a debugger. Currently Arm Development Studio and GDB are supported. During the test run, the script interacts with the debugger, sets breakpoints, triggers interrupts by writing into system registers, starts the target, and when the target is stopped, it examines the target's state.

A typical execution scenario looks like this:



Once started inside the debugger, the script automatically deduces the debugger it is running in, by trying to import the support libraries for a specific debugger. The order the debuggers are tried in the following order:

1. Arm Development studio
2. GDB

If both check fails, the script falls back to 'dummy' mode which means that the calls to the debugger log the call, and returns successfully.

Note: This 'dummy' mode can be used out of a debugger environment as well.

Important: The script assumes that the symbols for the software being debugged/tested are loaded in the debugger.

The available parameters are:

short option	long option	meaning
`-w`	`--sw-break`	Use sw breakpoint (the default is HW breakpoint)
`-q <IRQS>`	`--irqs <IRQS>`	The name of the IRQs json
`-t <TESTCASE>`	`--testcase <TESTCASE>`	The name of the file containing the testcase
`-b <BREAKPOINTS>`	`--breakpoints <BREAKPOINTS>`	The name of the breakpoints json file

5.3 Input files

5.3.1 Breakpoints

below is a sample file for breakpoints:

```
{
    "breakpoints": {
        "irq_test_iteration_before_service_calls": {
            "file": "core_ns_positive_testsuite.c",
            "line": 692
        },
        "irq_test_service1_high_handler": {
            "symbol": "SPM_CORE_IRQ_TEST_1_SIGNAL_HIGH_isr"
        },
        "irq_test_service2_prepare_veneer": {
            "offset": "4",
            "symbol": "tfm_spm_irq_test_2_prepare_test_scenario_veneer"
        }
    }
}
```

Each point where a breakpoint is to be set by the tool should be enumerated in this file, in the "breakpoints" object. For each breakpoint an object needs to be created. The name of the object can be used in the testcase description. The possible fields for a breakpoint object can be seen in the example above.

tools/generate_breakpoints.py

This script helps to automate the generation of the breakpoints from source files. Each code location that is to be used in a testcase, should be annotated with one of the following macro in the source files:

```
/* Put breakpoint on the address of the symbol */
#define IRQ_TEST_TOOL_SYMBOL(name, symbol)

/* Put a breakpoint on the address symbol + offset */
#define IRQ_TEST_TOOL_SYMBOL_OFFSET(name, symbol, offset)

/* Put a breakpoint at the specific location in the code where the macro is
 * called. This creates a file + line type breakpoint
```

(continues on next page)

(continued from previous page)

```
 */
#define IRQ_TEST_TOOL_CODE_LOCATION(name)
```

Usage of the script:

```
$ python3 generate_breakpoints.py --help
usage: generate_breakpoints.py [-h] tfm_source outfile

positional arguments:
tfm_source  path to the TF-M source code
outfile     The output json file with the breakpoints

optional arguments:
-h, --help  show this help message and exit
```

5.3.2 IRQs

```
{
    "irqs": {
        "test_service1_low": {
            "line_num" : 51
        },
        "ns_irq_low": {
            "line_num" : 40
        }
    }
}
```

Each IRQ that is to be triggered should have an object created inside the “irqs” object. The name of these objects is the name that could be used in a testcase description. The only valid field of the IRQ objects is “line_num” which refers to the number of the interrupt line.

5.3.3 Testcase

```
{
    "description" : ["Trigger Non-Secure interrupt during SPM execution in",
                     "privileged mode"],
    "steps": [
        {
            "wait_for" : "irq_test_iteration_start"
        },
        {
            "wait_for" : "spm_partition_start"
        },
        {
            "description" : ["Trigger the interrupt, but expect the operation",
                             "to be finished before the handler is called"],
            "expect" : "spm_partition_start_ret_success",
            "trigger" : "ns_irq_low"
```

(continues on next page)

(continued from previous page)

```
        },
        {
            "wait_for" : "ns_irq_low_handler"
        },
        {
            "wait_for" : "irq_test_service2_prepare"
        }
    ]
}
```

The test is executed by the script on a step by step basis. When the script is started, it processes the first step, then starts the target. After a breakpoint is hit, it processes the next target, and continues. This iteration is repeated until all the steps are processed

For each step, the following activities are executed:

1. All the breakpoints are cleared in the debugger
2. If there is a 'wait_for' field, a breakpoint is set for the location specified.
3. If there is a 'trigger' field, an IRQ is pended by writing to NVIC registers.
4. If there is an 'expect' field, a breakpoint is set for the location specified. Then the testcase file is scanned starting with the next step, and a breakpoint is set at the first location specified with a 'wait_for' field. Next time, when the execution is stopped, the breakpoint that was hit is compared to the expected breakpoint.

Each object can have a description field to add comments.

5.4 How to run the example

Before running the example, the breakpoints.json needs to be generated from the TF-M source tree:

```
$ cd tools/irq_test/
$ python3 tools/generate_breakpoints.py ../.. example/breakpoints.json
```

The example also require the regression test suite being present in the TF-M binary, so either `ConfigRegressionIPC.cmake` or `ConfigRegression.cmake` have to be used to compile TF-M. Also `-DCMAKE_BUILD_TYPE=Debug` config option have to be used in the cmake generation command, to be sure that the debug information is generated in the axf files.

The sequence of running the testcase in the `example` folder looks like the following:

1. Check out a version of TF-M that contains the `IRQ_TEST_TOOL_*` macros for the testcase
2. Generate breakpoints.json using the TF-M working copy above
3. Build TF-M checked out above
4. Start the debugger, connect to the target, and stop the target. (Make sure that the target is stopped before the IRQ testcase of the positive core test suite in TF-M starts executing, as the IRQ test tool's testcase uses the entry of that TF-M test as a trigger to start.)
5. Execute the script. The script automatically sets the breakpoint for the first step of the testcase, and continues the target execution.
6. Examine the output of the script. Successful execution is signalled by the following output line:

```
===== INFO: All the steps in the test file are executed successfully with the␣
↪expected result.
```

5.4.1 Arm Development Studio

The script can be called directly from the debugger's command window:

Note: In the command absolute path have to be used both for the `irq_test.py` and for the parameters.

```
source irq_test.py -q example/irqs_AN521.json -b example/breakpoints.json -t example/
↪testcase.json
```

5.4.2 GDB

The script should be sourced inside GDB, without passing any arguments to it.

```
(gdb) source irq_test.py
```

That registers a custom command `test_irq`. `test_irq` should be called with three parameters: breakpoints, irqs, and the test file. This command will actually execute the tests.

Note: This indirection in case of GDB is necessary because it is not possible to pass parameters to the script when it is sourced.

Important: The script needs to be run from the <TF-M root>/tools/irq_test directory as the 'current working dir' is added as module search path.

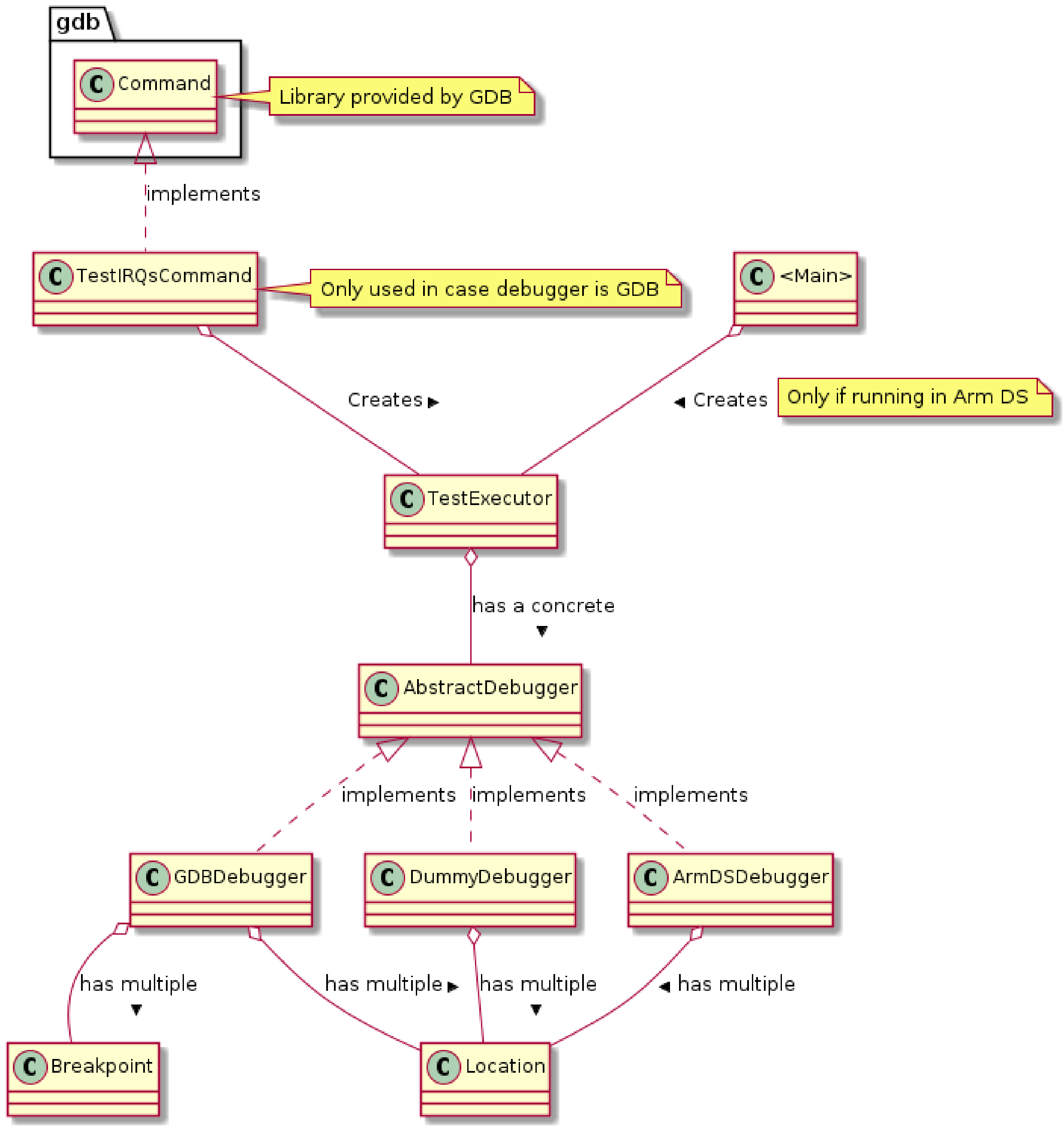
A typical execution of the script in GDB would look like the following:

```
(gdb) target remote localhost: 2331
(gdb) add-symbol-file /path/to/binaries/tfm_s.axf 0x10000000
(gdb) add-symbol-file /path/to/binaries/tfm_ns.axf 0x00040000
(gdb) source /path/to/script/irq_test.py
(gdb) test_irq -q example/irqs_LPC55S69.json -b example/breakpoints.json -t example/
↪testcase.json
```

Note: `add-symbol-file` command is used above as other commands like `file` and `symbol-file` seem to be dropping the previously loaded symbols. The addresses the axf files are loaded at are depending on the platform they are built to. The address needs to be specified is the start of the code section

5.5 Implementation details

Class hierarchy:



5.6 Possible further improvements

- Add priority property to the IRQs data file
- Add possibility to run randomized scenarios, to realise stress testing.

CHAPTER

SIX

LIBRARY DEPENDENCY TRACE TOOL

Table of Contents

- *Library dependency trace tool*
  - *Install graphviz*
    - *Linux*
    - *Windows*
  - *Usage*
    - *Dependency trace output diagram*
    - *Dependency trace output log*

This is a script to search a specific library's dependencies, including linked libraries, source files, include paths and some other information about it.

6.1 Install graphviz

6.1.1 Linux

Install graphviz to generate diagram of dependencies.

```
sudo apt install graphviz # via Ubuntu package management
pip install graphviz # via Python package installer
```

6.1.2 Windows

In windows, graphviz is also needed and moreover, it is required to install the software with same name. It can be downloaded from graphviz. Note that it shall be added into system path of Windows.

6.2 Usage

The command is:

```
python3 lib_trace.py -l <library_name>
                     -p <repo_path_1,repo_path_2,...>
                     -d <max depth>
                     -i # only draw input dependent libraries
                     -o # only draw output dependent libraries
                     -h # help
```

It is required to input library name and repo paths, and at least one of `-o` or `-i`. The default max depth of graph is 1. The output files are `<library>.png`, `<library>.log` and `<library>.txt`. The `<library>.png` shows the relationship of the specific library. The `<library>.log` gives the information of the library with JSON format. The `<library>.txt` gives plain text format of the information.

6.2.1 Dependency trace output diagram

```
python3 lib_trace.py -p '<tf-m-path>,<tf-m-tests-path>' \
-l secure_fw -d 2 -i
```

From the piture we can see the input dependent libraries of `secure_fw`. The edges in the diagram show like:

```
 ------          condition           ------
|source|  ----------------------->  |target|
 ------                              ------
```

In CMAKE, it means:

```
target_link_libraries(target
    PUBLIC|PRIVATE|INTERFACE
        <condition> : source
)
```

Different CMAKE link key word shows different edge color:

CMAKE link key words	Edge Colors
PUBLIC	Green
PRIVATE	Red
INTERFACE	Blue

Different node of library or target has different color:

Node meaning	Node Colors
Static library	Green
Interface library	Light blue
Unkown from workspace	Pink

Note: The pink node is an exception, it can be a static, interface library. It can be an executable target in CMAKE as well. This tool cannot locate where it is created if the workspace isn't added into the list of imput paths.

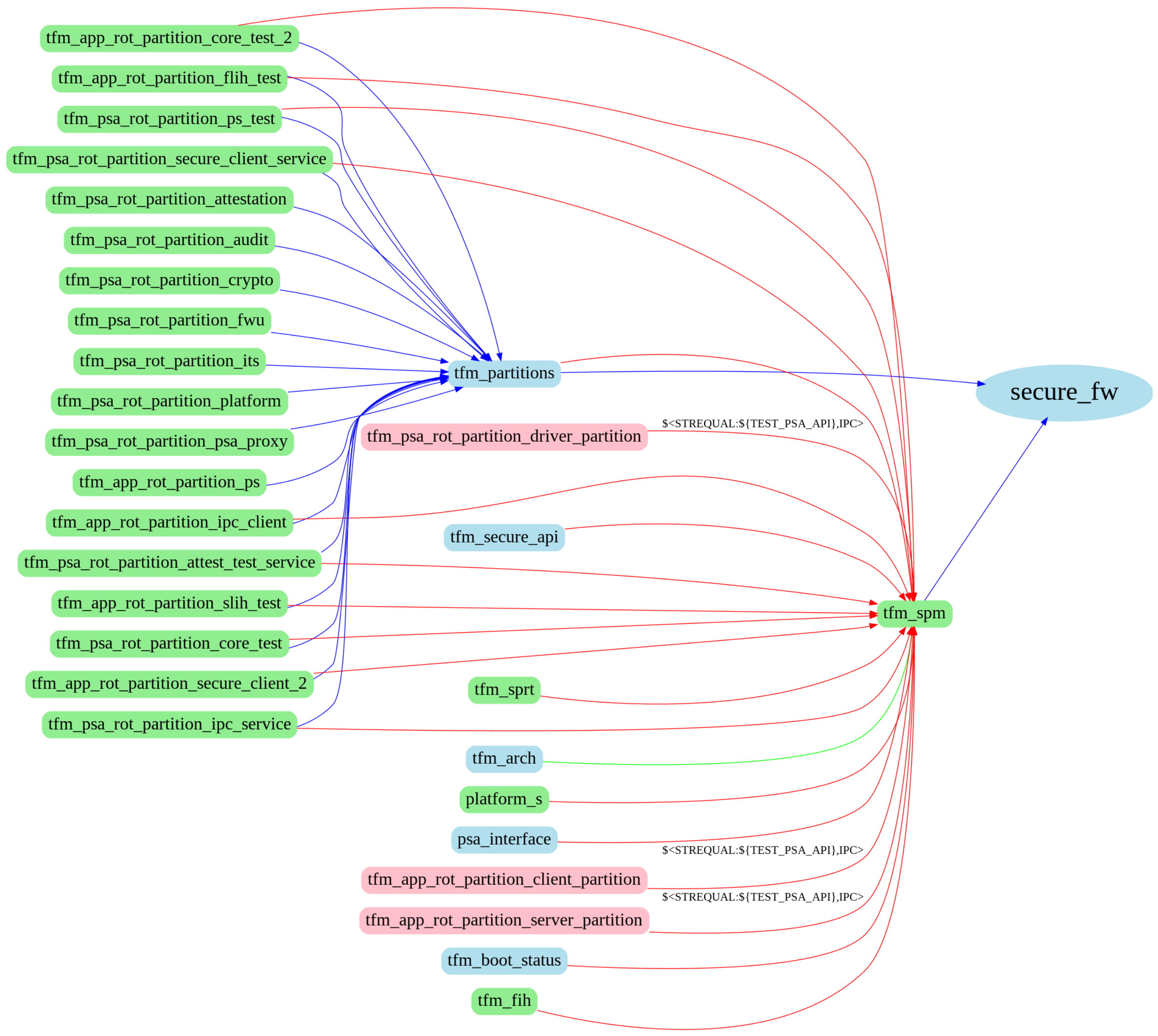

tfm_app_rot_partition_core_test_2
tfm_app_rot_partition_flih_test
tfm_psa_rot_partition_ps_test
tfm_psa_rot_partition_secure_client_service
tfm_psa_rot_partition_attestation
tfm_psa_rot_partition_audit
tfm_psa_rot_partition_crypto
tfm_psa_rot_partition_fwu
tfm_psa_rot_partition_its
tfm_psa_rot_partition_platform
tfm_psa_rot_partition_psa_proxy
tfm_app_rot_partition_ps
tfm_app_rot_partition_ipc_client
tfm_psa_rot_partition_attest_test_service
tfm_app_rot_partition_slih_test
tfm_psa_rot_partition_core_test
tfm_app_rot_partition_secure_client_2
tfm_psa_rot_partition_ipc_service
tfm_partitions
tfm_psa_rot_partition_driver_partition
$<STREQUAL:${TEST_PSA_API},IPC>
tfm_secure_api
tfm_sprt
tfm_arch
platform_s
psa_interface
$<STREQUAL:${TEST_PSA_API},IPC>
tfm_app_rot_partition_client_partition
$<STREQUAL:${TEST_PSA_API},IPC>
tfm_app_rot_partition_server_partition
tfm_boot_status
tfm_fih
tfm_spm
secure_fw

Figure 1:: Library secure_fw's dependent libraries

Another diagram of `tfm_sprt` shows:

```
python3 lib_trace.py -p '<tf-m-path>,<tf-m-tests-path>' \
-l tfm_sprt -d 1 -i -o
```

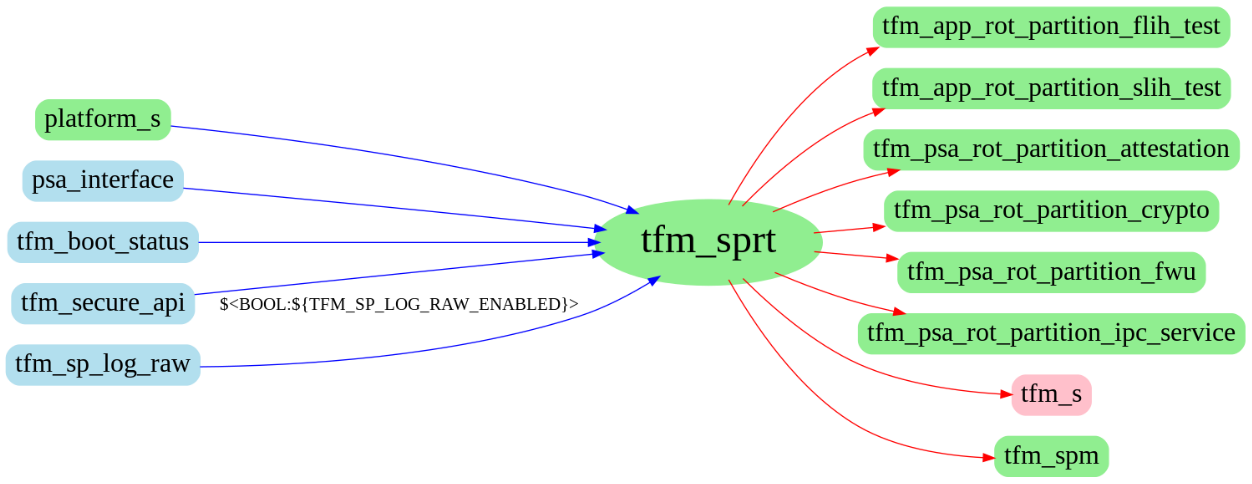

platform_s
psa_interface
tfm_boot_status
tfm_secure_api
tfm_sp_log_raw
$<BOOL:${TFM_SP_LOG_RAW_ENABLED}>
tfm_sprt
tfm_app_rot_partition_flih_test
tfm_app_rot_partition_slih_test
tfm_psa_rot_partition_attestation
tfm_psa_rot_partition_crypto
tfm_psa_rot_partition_fwu
tfm_psa_rot_partition_ipc_service
tfm_s
tfm_spm

Figure 2:: Library tfm_sprt's dependency trace

This picture shows the specific library's input and output dependencies.

6.2.2 Dependency trace output log

The log shows in two ways, a plain text or JSON file. The plain text is more readable. The text behind `<----` is the condition if the source has.

```
library name: library name

source libraries:
    INTERFACE
        tfm_partitions
        tfm_spm

destination libraries:
    PRIVATE
        tfm_app_rot_partition_ps
        tfm_psa_rot_partition_attestation
        tfm_psa_rot_partition_psa_proxy
        tfm_s

include directories:
    INTERFACE
        /home/sjl/work/tf-m/secure_fw/include       <----   BUILD_INTERFACE
        /home/sjl/work/tf-m/secure_fw/partitions    <----   BUILD_INTERFACE

source files:

compiler definitions:
```

The JSON file is supplied for users to get formatted input.

```
{
    "library name": "secure_fw",
    "source library": {
        "PUBLIC": [],
        "PRIVATE": [],
        "INTERFACE": [
            {
                "name": "tfm_partitions",
                "condition": ""
            },
            {
                "name": "tfm_spm",
                "condition": ""
            }
        ]
    },
    "dst library": {
        "PUBLIC": [],
        "PRIVATE": [
            {
                "name": "tfm_app_rot_partition_ps",
                "condition": ""
            },
            {
                "name": "tfm_psa_rot_partition_attestation",
                "condition": ""
            },
            {
                "name": "tfm_psa_rot_partition_psa_proxy",
                "condition": ""
            },
            {
                "name": "tfm_s",
                "condition": ""
            }
        ],
        "INTERFACE": []
    },
    "include": {
        "PUBLIC": [],
        "PRIVATE": [],
        "INTERFACE": [
            {
                "name": "/home/sjl/work/tf-m/secure_fw/include",
                "condition": "BUILD_INTERFACE"
            },
            {
                "name": "/home/sjl/work/tf-m/secure_fw/partitions",
                "condition": "BUILD_INTERFACE"
            }
        ]
```

(continues on next page)

(continued from previous page)

```
    },
    "source": {
        "PUBLIC": [],
        "PRIVATE": [],
        "INTERFACE": []
    },
    "compile definition": {
        "PUBLIC": [],
        "PRIVATE": [],
        "INTERFACE": []
    }
}
```

CHAPTER

SEVEN

PROFILER TOOL AND TF-M PROFILING

The profiler is a tool for profiling and benchmarking programs. The developer can leverage it to get the interested data of runtime.

Initially, the profiler supports only count logging. You can add "checkpoint" in the program. The timer count or CPU cycle count of this checkpoint can be saved at runtime and be analysed in the future.

7.1 TF-M Profiling Build Instructions

TF-M has integrated some built-in profiling cases. There are two configurations for profiling:

- `CONFIG_TFM_ENABLE_PROFILING`: Enable profiling building in TF-M SPE and NSPE. It cannot be enabled together with any regression test configs, for example `TEST_NS`.
- `TFM_TOOLS_PATH`: Path of tf-m-tools repo. The default value is `DOWNLOAD` to fetch the remote source.

The section *TF-M Profiling Cases* introduces the profiling cases in TF-M. To enable the built-in profiling cases in TF-M, run:

```
cd <path to tf-m-tools>/profiling/profiling_cases/tfm_profiling
mkdir build

# Build SPE
cmake -S <path to tf-m> -B build/spe -DTFM_PLATFORM=arm/mps2/an521 \
      -DCONFIG_TFM_ENABLE_PROFILING=ON -DCMAKE_BUILD_TYPE=Release \
      -DTFM_EXTRA_PARTITION_PATHS=${PWD}/../prof_psa_client_api/partitions/prof_server_
↪partition;${PWD}/../prof_psa_client_api/partitions/prof_client_partition \
      -DTFM_EXTRA_MANIFEST_LIST_FILES=${PWD}/../prof_psa_client_api/partitions/prof_psa_
↪client_api_manifest_list.yaml \
      -DTFM_PARTITION_LOG_LEVEL=TFM_PARTITION_LOG_LEVEL_INFO

# Another simple way to configure SPE:
cmake -S <path to tf-m> -B build/spe -DTFM_PLATFORM=arm/mps2/an521 \
      -DTFM_EXTRA_CONFIG_PATH=${PWD}/../prof_psa_client_api/partitions/config_spe.cmake
cmake --build build/spe -- install -j

# Build NSPE
cmake -S . -B build/nspe -DCONFIG_SPE_PATH=${PWD}/build/spe/api_ns \
      -DTFM_TOOLCHAIN_FILE=build/spe/api_ns/cmake/toolchain_ns_GNUARM.cmake
cmake --build build/nspe -- -j
```

Note: TF-M profiling implementation relies on the physical CPU cycles provided by hardware timer (refer to *Implement the HAL*). It may not be supported on virtual platforms or emulators.

7.2 Profiler Integration Reference

profiler/profiler.c is the main source file to be complied with the tagert program.

7.2.1 Initialization

`PROFILING_INIT()` defined in *profiling/export/prof_intf_s.h* shall be called on the secure side before calling any other API of the profiler. It initializes the HAL and the backend database which can be customized by users.

Implement the HAL

export/prof_hal.h defines the HAL that should be implemented by the platform.

- `prof_hal_init()`: Initialize the counter hardware.
- `prof_hal_get_count()`: Get current counter value.

Users shall implement platform-specific hardware support in `prof_hal_init()` and `prof_hal_get_count()` under *export/platform*.

Take *export/platform/tfm_hal_dwt_prof.c* as an example, it uses Data Watchpoint and Trace unit (DWT) to count the CPU cycles which can be a reference for performance.

Setup Database

The size of the database is determined by `PROF_DB_MAX` defined in *export/prof_common.h*.

The developer can override the size by redefining `PROF_DB_MAX`.

7.2.2 Add Checkpoints

The developer should identify the places in the source code for adding the checkpoints. The count value of the timer or CPU cycle will be saved into the database for the checkpoints. The interface APIs are defined in *export/prof_intf_s.h* for the secure side.

It's also supported to add checkpoints on the non-secure side. Add *export/ns/prof_intf_ns.c* to the source file list of the non-secure side. The interface APIs for the non-secure side are defined in *export/ns/prof_intf_ns.h*.

The counter logging related APIs are defined in macros to keep the interface consistent between the secure and non-secure sides.

Users can call macro `PROF_TIMING_LOG()` logs the counter value.

```
PROF_TIMING_LOG(topic_id, cp_id);
```

Parameters	Description
topic_id	Topic is used to gather a group of checkpoints. It's useful when you have many checkpoints for different purposes. Topic can help to organize them and filter the related information out. It's an 8-bit unsigned value.
cp_id	Checkpoint ID. Different topics can have same cp_id. It's a 16-bit unsigned value.

7.2.3 Collect Data

After successfully running the program, the data should be saved into the database. The developer can dump the data through the interface defined in the header files mentioned above.

For the same consistent reason as counter logging, the same macros are defined as the interfaces for both secure and non-secure sides.

The data fetching interfaces work in a stream way. `PROF_FETCH_DATA_START` and `PROF_FETCH_DATA_BY_TOPIC_START` search the data that matches the given pattern from the beginning of the database. `PROF_FETCH_DATA_CONTINUE` and `PROF_FETCH_DATA_BY_TOPIC_CONTINUE` search from the next data set of the previous result.

Note: All the APIs increase the internal search index, be careful about mixing using them for different checkpoints and topics at the same time.

The match condition of a search is controlled by the tag mask. It's `tag value & tag_mask == tag_pattern`. To enumerate the whole database, set `tag_mask` and `tag_pattern` both to `0`.

- `PROF_FETCH_DATA_XXX`: The generic interface for getting data.
- `PROF_FETCH_DATA_BY_TOPIC_XXX`: Get data for a specific `topic`.

The APIs return `false` if no matching data is found until the end of the database.

7.2.4 Calibration

The profiler itself has the tick or cycle cost. To get more accurate data, a calibration system is introduced. It's optional.

The counter logging APIs can be called from the secure or non-secure side. And the cost of calling functions from these two worlds is different. So, secure and non-secure have different calibration data.

The system performance might float during the initialization, for example, change CPU frequency, enable cache, etc. So, it's recommended that the calibration is done just before the first checkpoint.

- `PROF_DO_CALIBRATE`: Call this macro to get the calibration value. The more `rounds` the more accurate.
- `PROF_GET_CALI_VALUE_FROM_TAG`: Get the calibration value from the tag. The calibrated counter is `current_counter - previous_counter - current_cali_value`. Here `current_cali_value` equals `PROF_GET_CALI_VALUE_FROM_TAG` (current_tag).

7.2.5 Data Analysis

Data analysis interfaces can be used to do some basic analysis and the data returned is calibrated already.

`PROF_DATA_DIFF`: Get the counter value difference for the two tags. Returning `0` indicates errors.

If the checkpoints are logged by multi-times, you can get the following counter value differences between two tags:

- `PROF_DATA_DIFF_MIN`: Get the minimum counter value difference for the two tags. Returning `UINT32_MAX` indicates errors.
- `PROF_DATA_DIFF_MAX`: Get the maximum counter value difference for the two tags. Returning `0` indicates errors.
- `PROF_DATA_DIFF_AVG`: Get the average counter value difference for the two tags. Returning `0` indicates errors.

A customized software or tool can be used to generate the analysis report based on the data.

7.2.6 Profiler Self-test

profiler_self_test is a quick test for all interfaces above. To build and run in the Linux:

```
cd profiler_self_test
mkdir build && cd build
cmake .. && make
./prof_self_test
```

7.3 TF-M Profiling Cases

The profiler tool has already been integrated into TF-M to analyze the program performance with the built-in profiling cases. Users can also add a new profiling case to get a specific profiling report. TF-M profiling provides example profiling cases in *profiling_cases*.

7.3.1 PSA Client API Profiling

This profiling case analyzes the performance of PSA Client APIs called from SPE and NSPE, including `psa_connect()`, `psa_call()`, `psa_close()` and `stateless psa_call()`. The main structure is:

```
prof_psa_client_api/
    ├── cases
    │     ├── non_secure
    │     └── secure
    └── partitions
          ├── prof_server_partition
          └── prof_client_partition
```

- The *cases* folder is the basic SPE and NSPE profiling log and analysis code.
- NSPE can use *prof_log* library to print the analysis result.
- *prof_server_partition* is a dummy secure partition. It immediately returns once it receives a PSA client call from a client.
- *prof_client_partition* is the SPE profiling entry to trigger the secure profiling.

To make this profiling report more accurate, It is recommended to disable other partitions and all irrelevant tests.

7.3.2 Adding New TF-M Profiling Case

Users can add source folder *<prof_example>* under path *profiling_cases* to customize performance analysis of target processes, such as the APIs of secure partitions, the functions in the SPM, or the user's interfaces. The integration requires these steps:

1. Confirm the target process block to create profiling cases.
2. Enable or create the server partition if necessary. Note that the other irrelevant partitions shall be disabled.
3. Find ways to output profiling data.
4. Trigger profiling cases in SPE or NSPE.
   a. For SPE, a secure client partition can be created to trigger the secure profiling.
   b. For NSPE, the profiling case entry can be added to the 'tfm_ns' target under the *tfm_profiling* folder.

Note: If the profiling case requires extra out-of-tree secure partition build, the paths of extra partitions and manifest list file shall be appended in `TFM_EXTRA_PARTITION_PATHS` and `TFM_EXTRA_MANIFEST_LIST_FILES`. Refer to Adding Secure Partition.

CHAPTER

EIGHT

SQUAD METRICS DASHBOARD

Author
Hugo L'Hostis

Organization
Arm Limited

Contact
hugo.lhostis@arm.com

8.1 SQUAD Presentation

Software Quality Dashboard (SQUAD) is a tool used by Trusted Firmware-M (TF-M) to keep track of some metrics for the project. It is a Linaro project (see here a link to the SQUAD Documentation).

For TF-M the purpose of having such a tool available is to have a history on some metrics of the project for different configurations of TF-M.

The TF-M SQUAD is available here : TFM's SQUAD. There are several configurations and metrics that can be selected, here is a link with some of them already selected : SQUAD parametered for Default, profileS and MinsizeProfileS.

8.2 Metrics and configurations

The metrics sent to the dashboard are currently the memory footprint measurements from the TFM project calculated by the arm-none-eabi-size function for the 2 files bl2.axf and tfm_s.axf :

- Text section size.
- bss section size.
- data total size.
- Total file size.

Each metric sent to the dashboard can be compared for different configurations used. The configurations available currently are the following :

- Default
- CoreIPC
- CoreIPCTfmLevel2
- DefaultProfileS
- DefaultProfileM

- DefaultProfileL
- MinSizeProfileS (DefaultProfileS with MinsizeRel Cmake type)

For all of the configurations, tests are disabled and the release build type is used. More configurations and metrics could be added to the dashboard in the future.

For each metric sent to the SQUAD dashboard, the metric must be linked with the state of the TF-M repository at the time of the build. On the dashboard this is visible from a string of numbers in the abscissa of the graphs displayed in SQUAD. This is the change ID of the latest commit used for the build. This means that in the case of 2 consecutive days with no new commit for TF-M, no new data points will be created.

8.3 CI integration

The data is currently sent by the TFM CI's nightly build.

The parameter "SQUAD_CONFIGURATIONS" of the CI build is what will trigger the configurations sent to the SQUAD dashboard. It should be set to the configurations's names separated by a comma, example :

```
SQUAD_CONFIGURATIONS = Default,DefaultProfileS
```

In this case, the 2 configurations Default and DefaultProfileS will be enabled and the data for those (and only those 2) will be sent to the dashboard. This is case insensitive and will ignore any space between the configurations. All the files manipulating the data and sending the data is available in the tf-m-ci-scripts repo.

The script memory_footprint.py is launched by the CI nightly build, it gathers and sends the data.

8.4 Adding a new platform

Currently, all the data sent is from AN521 builds. To add a new platform to the dashboard :

1. If the new platform is not already tested by the CI nightly build, it needs to be added.

2. The `memory_footprint.py` file in the tf-m-ci-scripts repo has to be modified to recognise the new platform.

3. If `memory_footprint.py` detects the new platform with a reference configuration, it should send a new metric with a different name to the existing ones containing the platform's name.

The data will then be accessible by selecting the correct metrics on the dashboard.

CHAPTER

NINE

STATIC CHECKING FRAMEWORK

This tool has been made to provide a framework to check the truster-firmware-m (TF-M) code base.

9.1 Instructions

This tool should be used from the root of the TF-M repository. launching run_all_checks.sh will launch the different checkers used :

- *Cppcheck*
- *Copyright header check*
- *clang-format*
- *Checkpatch*

Both checkpatch and clang-format are use to check the coding style, but they both cover different cases so together they provide a better coverage.

Each tool will be configured using a setup.sh script located under the tool directory before being launched. The main script might need to be launched with root priviledges in order to correctly install the tools on the first time it is being used.

The tool will return exit code of 0 if everything is compliant, and no new warnings are generated, and 1 in all other occasions.

Output reports if produced by each corresponding script, will be stored at *{TFM-Root}/checks_reports*`

9.1.1 Cppcheck

cppcheck is a tool used to check a number of checks on the codebase. The list of all the checks is available at : https://sourceforge.net/p/cppcheck/wiki/ListOfChecks/

tool configuration

This tool is using the pre-existing cppcheck configurations (arm-cortex-m.cfg/tfm-suppress-list.txt), implementing the developer's guidelines, as used by the TF-M CI.

The suppression list contains:

- Files that are not guaranteed to comply with the TF-M rules.
- Files under directories that correspond to external libraries.

The files utils.sh, arm-cortex-cfg.cfg and tfm-suppress-list.txt were copied from the CI scripts repo : https://review.trustedfirmware.org/admin/repos/ci/tf-m-ci-scripts

Using this tool

This script must be launched from the TFM repo and the reports will be stored under checks_reports if the xml option is selected. The possible parameters are the following:

- '-h' display the help for this tool
- '-r' select the raw output option. If this parameter is selected, the output will be displayed in the console instead of stored in an xml file

---

9.1.2 clang-format

This tool uses clang-format and the script clang-format-diff.py provided in the clang tools to apply style checking on all staged files. You can use it before committing to check for any style that does not comply with the TF-M coding guidelines.

How to use it

Using the configuration script

- Install clang-format on your system, the tool is available as part of the clang suite, but can also be installed standalone from a packet manager.
- After that the only thing the user needs to do is run the run-clang-format.sh script while in the TF-M root folder. If any dependency is missing, it will call the setup script to install it.

Without using the configuration script

- Make sure clang-format is installed in your system
- Copy the .clang-format file to the root of the tf-m directory -
- Download clang-format-diff from the llvm github repository
- Run

```
git diff -U0 --no-color HEAD^ | <path/to/clang-format-diff.py> -p1 -style file> <out_
↪file>
```

If clang-format makes any correction, a diff file will be created in the tfm root directory, simply run `git apply -p0 <diff_file>` to apply them. The generated diff is a unified diff, whose format is slightly different that the git diff format, hence the -p0 option so that git can correctly interpret the file.

9.1.3 Checkpatch

This tool uses the checkpatch tool, with a prexisting configuration, created for the TF-M OpenCI, to perform static checking on the developer's machine.

The script is set by default, to mimic the configuratio of TF-M OpenCI script, but the user can extend it by adding new parameters in the `checkpatch.conf` file.

The script is kept simple by design, since it is aimed at be easy to maintain from a single's user perspective

Set-up

Setting up this tool, is a simple operation of retrieving the script and its dependencies and placing them in the *tf-m-toolsstatic_checkscheckpatch* directory:

- checkpatch.pl
- const_structs.checkpatch
- spelling.txt

The proccess can be automated, without any special priviledges by invoking the `tf-m-tools\static_checks\checkpatch\setup.sh` script.

Using the script

The user can call the `tf-m-tools\static_checks\checkpatch\run_checkpatch.sh` script from the `{$TFM-ROOT}` directory

```
cd $TFM-ROOT
# Only need to be run once
../tf-m-tools/static_checks/checkpatch/setup.sh
../tf-m-tools/static_checks/checkpatch/run_checkpatch.sh
```

Or as a part of all the tests set in the Static Checking Framework

```
cd $TFM-ROOT
../tf-m-tools/static_checks/run_all_checks.sh
```

9.1.4 Copyright header checks

This script checks that all text files staged for commit (new and modified) have the correct license header. It returns the list of files whose header is missing or not updated. To use it, make sure you have jinja2 installed (if you are on linux you can run the setup.sh script to install it), then run the python script from the tfm repository with the name of the organization, for example: `python3 run_header_check.py Arm` To get the list of known organizations, run `python3 run_header_check.py --help`.

The list is stored in a python file called "orgs_list.py", stored in the same directory as the script. To add a new organization, add a generic name and the official denomination used in the copyright header to this file.

The copyright header must have the following structure: Copyright (c) <year>, <organisation>. (optional)All rights reserved.

9.1.5 Git Hooks

This directory is aimed at providing **code snippets** which can be inserted to developer's git hooks, and allow them to run the Static Check Framework before pushing a patch.

For full description and capabilities, please refer to the official documentation for Git SCM Hooks .

Adding a hook

To use a specific hook, please manually copy the snippet to the file with matching name, located at

```
TF-M-ROOT/.git/hooks/{HOOK_FILENAME}
```

Pre-Push

Enabling SCF requires adding the snippet on the pre-push hook. This is so that the check can be performed **BEFORE** a developer pushes a new patch to Gerrit and *ONLY IF* requested by the user.

With the aim of making the functionality unintrusive, the following environment variables are required.

- *SCF_ORG*: To set the Organisation for the purposes of Header/Copyright check.
- *SCF_ENABLE*: To enable checking before pushing a patch.

_If not set SCF_ORG defaults to 'arm'_

Custom directory paths

By default the reference code assumes the standard directory structure for TF-M and dependencies.

```
└── dev-dir
    ├── tf-m
    ├── tf-m-tools
```

The user can override this by setting the *TFM_ROOT_PATH*, *SCF_TEST_PATH*, *SCF_TEST_APP* environment variables, to match his needs.

Using SCP

Assuming the developer has already set-up trusted-firmware-m and tf-m-tools and has already copied the snippet over to *<TF-M-ROOT>/.git/hooks/pre-push*

```
cd <TF-M-ROOT>
env SCF_ENABLE=1 git push origin HEAD:refs/for/master
```

CHAPTER

TEN

TF_FUZZ

.../tf_fuzz directory contents:

assets calls demo parser tests regression backupStuff class_forwards.hpp lib README tf_fuzz.cpp utility boilerplate commands Makefile template tf_fuzz.hpp visualStudio

TF-Fuzz root directory.

TF-Fuzz is a TF-M fuzzing tool, at the PSA-call level. At the time of writing this at least, presentations available at:

```
https://www.trustedfirmware.org/docs/TF-M_Fuzzing_Tool_TFOrg.pdf
https://zoom.us/rec/share/1dxZcZit111IadadyFqFU7IoP5X5aaa8gXUdr_UInxmMbyLzEqEmXQdx79-
↪IWQ9p
```

(These presentation materials may not be viewable by all parties.)

To build TF-Fuzz, simply type "make" in this directory. Executable, called "tfz", is placed in this directory.

To run tfz, two environment variables must first be assigned. In bash syntax:

```
export TF_FUZZ_LIB_DIR=<path to this TF-M installation>/tf-m-tools/tf_fuzz/lib
export TF_FUZZ_BPLATE=tfm_boilerplate.txt
```

Examples of usage can be found in the demo directory.

To generate a testsuite for TF-M from a set of template files, use generate_test_suite.sh.

```
Usage: generate_test_suite.sh <template_dir> <suites_dir>

Where:
    template_dir: The directory containing template files for the
                  fuzzing tool
    suites_dir: The suites directory in the TF-M working copy.
                  i.e.: $TF-M_ROOT/test/suites
Example:
    cd tf-m-tools/tf_fuzz
    ./generate_test_suite.sh $TF-M_ROOT/../tf-m-tools/tf_fuzz/tests/  $TF-M_ROOT/../tf-m-
↪tests/test/suites/
```

After the test suite is generated, the new test suite needs to be added to the TF-M build by providing the following options to the CMake generate command (The path needs to be aligned with the test suite dir provided for the shell script above):

```
-DTFM_FUZZER_TOOL_TESTS=1
-DTFM_FUZZER_TOOL_TESTS_CMAKE_INC_PATH=$TF-M_ROOT/../tf-m-tests/test/suites/tf_fuzz
```

To help understand the code, below is a C++-class hierarchy used in this code base. They are explained further in the READMEs in their respective direc- tories, so the file names where the classes are defined is listed below (this, very roughly in order of functional interactions, of chronological usage during execution, and of most-to-least importance):

```
template_line                     ./template/template_line.hpp
    sst_template_line             ./template/template_line.hpp
        read_sst_template_line    ./template/sst_template_line.hpp
        remove_sst_template_line  ./template/sst_template_line.hpp
        set_sst_template_line     ./template/sst_template_line.hpp
    policy_template_line          ./template/template_line.hpp
        read_policy_template_line ./template/crypto_template_line.hpp
        set_policy_template_line  ./template/crypto_template_line.hpp
    key_template_line             ./template/template_line.hpp
        read_key_template_line    ./template/crypto_template_line.hpp
        remove_key_template_line  ./template/crypto_template_line.hpp
        set_key_template_line     ./template/crypto_template_line.hpp
    security_template_line        ./template/template_line.hpp
        security_hash_template_line  ./template/secure_template_line.hpp

psa_call                          ./calls/psa_call.hpp
    crypto_call                   ./calls/psa_call.hpp
        policy_call               ./calls/crypto_call.hpp
            policy_get_call       ./calls/crypto_call.hpp
            policy_set_call       ./calls/crypto_call.hpp
        key_call                  ./calls/crypto_call.hpp
            get_key_info_call     ./calls/crypto_call.hpp
            set_key_call          ./calls/crypto_call.hpp
            destroy_key_call      ./calls/crypto_call.hpp
    sst_call                      ./calls/psa_call.hpp
        sst_remove_call           ./calls/sst_call.hpp
        sst_get_call              ./calls/sst_call.hpp
        sst_set_call              ./calls/sst_call.hpp
    security_call                 ./calls/psa_call.hpp
        hash_call                 ./calls/security_call.hpp

boilerplate                       ./boilerplate/boilerplate.hpp

psa_asset                         ./assets/psa_asset.hpp
    crypto_asset                  ./assets/crypto_asset.hpp
        policy_asset              ./assets/crypto_asset.hpp
        key_asset                 ./assets/crypto_asset.hpp
    sst_asset                     ./assets/sst_asset.hpp

tf_fuzz_info                      ./tf_fuzz.hpp
```

(continues on next page)

(continued from previous page)

```
crc32                                ./utility/compute.hpp

gibberish                            ./utility/gibberish.hpp

expect_info                          ./utility/data_blocks.hpp
set_data_info                        ./utility/data_blocks.hpp
asset_name_id_info                   ./utility/data_blocks.hpp
```

There are currently two especially annoying warts on the design of TF-Fuzz:

- Need better management of variables in the generated code. Currently, for example, upon "read"ing a value from a PSA asset more than once, it creates a same-named (i.e., duplicate) variable for each such time, which is obviously not right.
- Upon adding the ability to do "do any N of these PSA calls at random," in hindsight, a fundamental flaw was uncovered in the top-level flow of how TF-Fuzz generates the code. High-level summary:
  - It should have completely distinct Parse, Simulate, then Code-generation stages.
  - Currently, the Parse and Simulate stages aren't really completely distinct, so there's a bunch of complicated Boolean flags traffic- copping between what in hindsight should be completely-separate Parse vs. Code-generation functionality. The function, interpret_template_line(), currently in …/tf_fuzz/parser/tf_fuzz_grammar.y (which may be moved to the its own file with randomize_template_lines()), has the lion's share of such Booleans, such as fill_in_template, create_call_bool, and create_asset_bool. The way it *should* work is:
  - The parser in …/tf_fuzz_grammar.y should generate an STL vector (or list) of psa_call-subclass "tracker" objects. It should not generate PSA-asset tracker objects (subclasses of psa_asset).
  - There should then be an organized Simulate stage, that sequences through the psa_call-subclass list, creating and accumulating/maintaining current state in psa_asset-subclass objects, using that current state to determine expected results of each PSA call, which get annotated back into the psa_call-tracker objects.
  - Finally, there already is, and should continue to be, a Code-generation phase that writes out the code, based upon text substitutions of "boilerplate" code snippets.
  - Currently, (hindsight obvious) the Parse and Simulate phases got somewhat muddled together. This shouldn't be super-hard to fix. That final Code-generation phase, conceptually at least, could be replaced instead with simply executing those commands directly, for targets that sufficient space to run TF-Fuzz in real-time.

CHAPTER

ELEVEN

USING DOCKER TO BUILD TF-M

This tool has been made to provide an easy way to build the trusted firmware m project without having to set up a whole working environment.The only tool necessary is docker.

11.1 Configuration Parameters

The config file /config/container_cfg is used to set up the tool. the following parameters are available :

- CORE_IMG_NAME : Name of the main docker image running GNUARM
- ARMCLANG_IMG_NAME : Name of the image using ARMCLANG. This image is based on the core image.
- DOCS_IMG_NAME : Name of the image used to build the documentation. This image is based on the core image.
- BUILD_DIR : Name of the directory where the build files will be generated. If your current TFM repository has a directory named the same, it will be deleted.
- DOCS_DIR : Name of the directory where the documentation files will be generated. If your current TFM repository has a directory named the same, it will be deleted.
- LOCAL_TFM_REPO : path to your local tfm repository. this parameter is mandatory
- PLATFORM : Name of the platform used for the TFM build.
- ADDITIONNAL_PARAMETERS (optionnal) : additionnal parameters for the TFM build.

11.2 Building an image

To build the docker images (TFM_Core, TFM_Armclang and TFM_documentation), launch the build_images.sh script.

11.3 Running an image

To launch a container, launch the corresponding run_xxx.sh script. Your local TFM repo will be mounted in the containerand the generated files will be available in your local TFM repo after the build.

To launch a container and build TFM manually, use the following command :

docker run -it –rm –user $(id -u):$(id -g) -v /etc/group:/etc/group:ro
-v /etc/passwd:/etc/passwd:ro -v /etc/shadow:/etc/shadow:ro -v $LOCAL_TFM_REPO:/opt/trusted-firmware-m –entrypoint /bin/bash $CORE_IMG_NAME

Note : Armclang currently uses the ARMLTD_LICENSE_FILE variable which should point to a license server.
