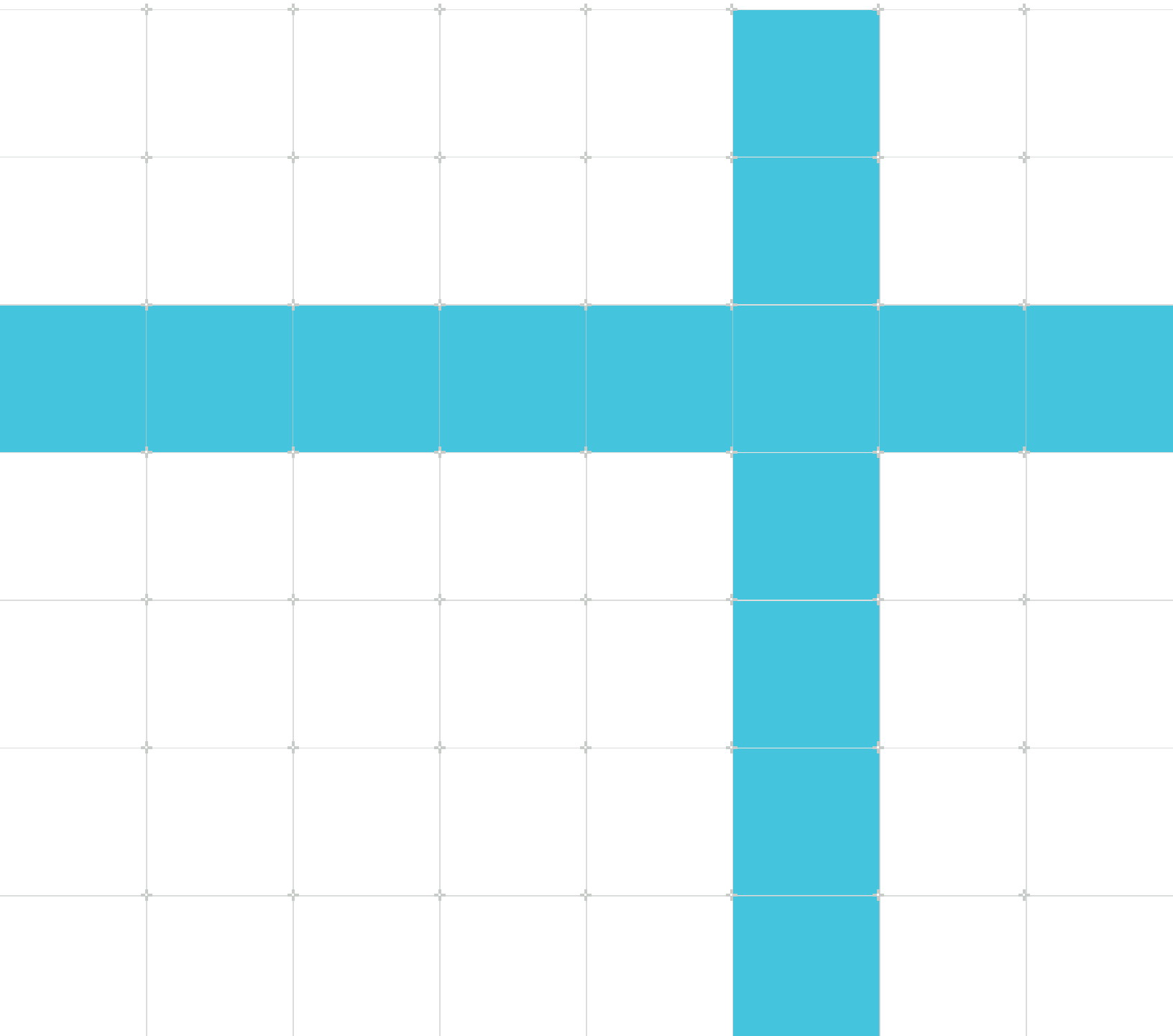
Arm® PSA-M Functional API Test Suite

Version 1.6

Validation Methodology

Non-Confidential

Issue 01
101447_0106_01_en



Arm® PSA-M Functional API Test Suite

Validation Methodology

Release Information

Document history

Issue	Date	Confidentiality	Change
A	28 September 2018	Non-Confidential	Alpha release
B	30 October 2018	Non-Confidential	Minor edits
C	15 January 2019	Non-Confidential	Beta release. The document number has been changed.
D	4 June 2019	Non-Confidential	Beta quality with minor updates
E	30 September 2019	Non-Confidential	Beta quality with minor updates
F	28 February 2020	Non-Confidential	EAC quality with minor updates
G	30 November 2020	Non-Confidential	EAC quality with minor updates
0102-01	30 June 2021	Non-Confidential	EAC release. The document now follows a new numbering format.
0103-01	8 October 2021	Non-Confidential	EAC release
0104-01	25 January 2022	Non-Confidential	EAC release
0105-01	31 May 2023	Non-Confidential	EAC release
0106-01	14 March 2024	Non-Confidential	EAC release

Proprietary Notice

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of this document shall prevail.

The validity, construction and performance of this notice shall be governed by English Law.

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

PRE-1121-V1.0

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by Arm and the party that Arm delivered this document to.

Unrestricted Access is an Arm internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com.

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

This document includes language that can be offensive. We will replace this language in a future issue of this document.

To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction

1.1 Conventions

The following subsections describe conventions used in Arm documents.

Glossary

The Arm® Glossary is a list of terms used in Arm documentation, together with definitions for those terms. The Arm Glossary does not contain terms that are industry standard unless the Arm meaning differs from the generally accepted meaning.

See the Arm Glossary for more information: developer.arm.com/glossary.

Convention	Use
italic	Citations.
bold	Terms in descriptive lists, where appropriate.
`monospace`	Text that you can enter at the keyboard, such as commands, file and program names, and source code.
`monospace underline`	A permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.
`<and>`	Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example: `MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>`
SMALL CAPITALS	Terms that have specific technical meanings as defined in the *Arm® Glossary*. For example, **IMPLEMENTATION DEFINED**, **IMPLEMENTATION SPECIFIC**, **UNKNOWN**, and **UNPREDICTABLE**.

We recommend the following. If you do not follow these recommendations your system might not work.

Your system requires the following. If you do not follow these requirements your system will not work.

You are at risk of causing permanent damage to your system or your equipment, or harming yourself.

This information is important and needs your attention.

A useful tip that might make it easier, better or faster to perform a task.

A reminder of something important that relates to the information you are reading.

1.2 Useful resources

This document contains information that is specific to this product. See the following resources for other useful information.

Access to Arm documents depends on their confidentiality:

- Non-Confidential documents are available at developer.arm.com/documentation. Each document link in the following tables goes to the online version of the document.
- Confidential documents are available to licensees only through the product package.

Arm product resources	Document ID	Confidentiality
Arm® Platform Security Architecture Firmware Framework specification	DEN0063	Non-Confidential
PSA Security model	DEN0128	Non-Confidential
Arm® Trusted Base System Architecture for Armv8-M	DEN0021F	Non-Confidential
Platform Security Boot Guide	DEN0072	Non-Confidential
PSA Cryptography API	IHI0086	Non-Confidential
Arm® v8 Architecture Reference Manual, Armv8 for M-profile	DDI0553B.q ID30092021	Non-Confidential
PSA Certified Secure Storage API 1.0	IHI0087	Non-Confidential
PSA Attestation API 1.0	IHI0085	Non-Confidential

Arm tests its PDFs only in Adobe Acrobat and Acrobat Reader. Arm cannot guarantee the quality of its documents when used with any other PDF reader.

Adobe PDF reader products can be downloaded at http://www.adobe.com.

1.3 Other information

See the Arm® website for other relevant information.

- Arm® Developer.
- Arm® Documentation.
- Technical Support.
- Arm® Glossary.

2. Introduction to PSA test suite

This chapter introduces the features and components of the functional API test suite for Arm® Firmware Framework for Armv8-M.

2.1 Abbreviations

This section lists the abbreviations used in this document.

Table 2-1: Abbreviations and expansions

Abbreviation	Expansion
API	Application Programming Interface
FF	Firmware Framework
ITS	Internal Trusted Storage
NSPE	Non-Secure Processing Element
PAL	Platform Abstraction Layer
PE	Processing Element
PS	Protected Storage
PSA	Platform Security Architecture
RoT	Root of Trust
RTOS	Real-Time Operating System
SPE	Secure Processing Element
SPM	Secure Partition Manager
SUT	System Under Test
VAL	Validation Abstraction Layer

2.2 PSA APIs

Arm® Platform Security Architecture (PSA) is a holistic set of threat models, security analyses, hardware and firmware architecture specifications, and an open-source firmware reference implementation.

PSA provides a recipe that allows security to be consistently designed at both hardware and firmware levels. One of the goals of PSA is to make IoT security easier and quicker. This means having reliable, consistent APIs, and useful built-in security functions for device manufacturers and the developer community. These PSA APIs provide a consistent developer experience, hiding the underlying complexity of the security system.

Arm PSA defines the following set of API specifications:

- PSA Firmware Framework

- PSA functional APIs

2.2.1 PSA Firmware Framework

PSA Firmware Framework (FF) defines a standard programming environment and firmware interfaces for implementing and accessing security services within Root of Trust (RoT) of a device.

PSA security model divides execution within the system into two domains:

- Non-secure Processing Environment (NSPE)
- Secure Processing Environment (SPE)

NSPE contains application firmware, and OS kernel and libraries. It controls most I/O peripherals. SPE contains security firmware and hardware resources that must be isolated from NSPE firmware and hardware resources. The security model requires that no NSPE firmware nor hardware can inspect or modify any SPE hardware, code, or data.

Security functionality is exposed by PSA as a collection of RoT services. Each RoT service is a set of related security functionality. For example, there may be an RoT service for cryptography operations, and another for Secure storage.

PSA subdivides the SPE into two subdomains:

- PSA RoT
- Application RoT

PSA RoT provides fundamental RoT services to the system and also manages the isolated execution environment for the application RoT services.

The following table describes the main components of PSA RoT.

Table 2-2: PSA RoT components

Component	Description
PSA security lifecycle	Identifies the production phase of the device and controls the availability of device secrets and sensitive capabilities such as Secure debug.
PSA immutable RoT	Hardware and non-modifiable firmware, and data installed during manufacturing.
Trusted boot and Firmware update	Ensures the integrity and authenticity of the device firmware.
Secure Partition Manager	Manages isolation of the RoT services, the IPC mechanism that allows software in one domain to make requests of another, and scheduling logic to ensure that requests are eventually serviced.
PSA RoT services	Provides essential cryptographic functionality and manage accesses to the immutable RoTs for application RoT services.

The Firmware Framework specification:

- Provides requirements for the Secure Partition Manager (SPM).
- Defines a standard runtime environment for developing protected RoT services, including the programming interfaces provided by the SPM for implementing and using RoT services.

- Defines the standard interfaces for the PSA RoT services.

For more information on SPM and PSA RoT, see the *Arm® Platform Security Architecture Firmware Framework specification*.

2.2.2 PSA functional APIs

PSA functional APIs are the top-level APIs used by application developers and Real-Time Operating System (RTOS) vendors. These APIs provide the top-level essential services related to Crypto, Secure storage, and attestation tokens.

These APIs have been designed for even non-expert software developers who want to implement hardware security features. For more information on PSA functional APIs, see the Functional APIs specification.

2.3 Test suite

Architecture tests are a set of examples of the invariant behaviors that are specified by the PSA API specifications. Use these tests to check if the behaviors are interpreted correctly in your system.

These tests cover checks for the following categories of features, each covering a different area of architecture.

Table 2-3: Test categories and their descriptions

API type	Test category	Subcategory	Description
PSA Firmware Framework	IPC	Level of isolation	Tests verifying the expected behavior of SPM involved in different levels of isolation, as defined by the specification.
		Client APIs	Tests that validate client APIs.
		Secure partition APIs	Tests that validate Secure partition APIs.
		Manifest input	Tests that validate manifest input parameters.
		PSA RoT lifecycle API	Tests that validate PSA RoT lifecycle API.
Functional APIs	Crypto	PSA Crypto APIs	Tests that validate PSA Crypto APIs.
	Internal Trusted Storage (ITS)	PSA ITS APIs	Tests that validate PSA ITS APIs.
	Protected Storage (PS)	PSA PS APIs	Tests that validate PSA PS APIs.
	Initial Attestation	PSA Initial Attestation API	Tests that validate PSA Initial Attestation API.

The test suite contains tests that have checks embedded within the test code. To view the list of test suites and how these different categories of features are checked, see test-list documents in the `docs/` directory.

2.4 Test suite components

The following table describes the test suite components.

Table 2-4: Test suite components

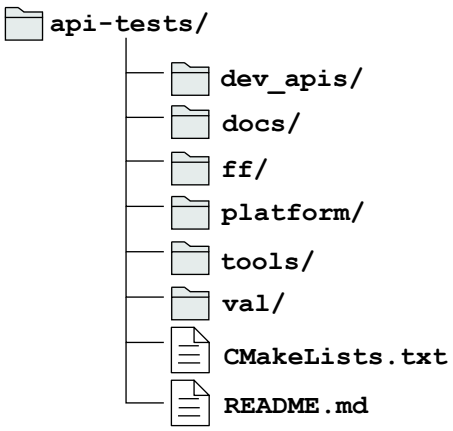
Component	Description
Test suites	Contains self-checking tests that are written in C.
Substructure	Test-supporting layers consist of a framework and libraries set up as: • Tools to build the tests • Validation Abstraction Layer (VAL) library • Platform Abstraction Layer (PAL) library
Documentation	Suite-specific documents such as test lists, porting guide, and API specification.

2.5 Directory structure

The test components must be in a specific hierarchy for the test suite.

The following figure contains the top-level directory files which is a release package downloaded from GitHub.

Figure 2-1: Test suite



dev_apis

has subsuites containing architecture tests for the functional APIs specification. This test suite is a set of C-based directed tests, each of which verifies the implementation against a test scenario that is described by the PSA functional APIs specification. These tests are abstracted from the underlying hardware platform by the VAL.

docs

contains the test suite documentation.

ff

has subsuites containing architecture tests for PSA-FF specification. This test suite is a set of C-based directed tests, each of which verifies the implementation against a test scenario that is described by the PSA-FF specifications. These tests are abstracted from the underlying hardware platform by the VAL.

platform

contains files to form the PAL. PAL is the closest to hardware and is aware of the underlying hardware details. Since this layer interacts with hardware, it must be ported or tailored to specific hardware required for system components present in a platform. This layer is also responsible for presenting a consistent interface to the VAL required for the tests.

tools

contains makefiles and scripts that are used to generate test binaries.

val

contains subdirectories for the VAL libraries. This layer provides a uniform and consistent view of the available test infrastructure to the tests in the test suite. VAL makes appropriate calls to the PAL to achieve this functionality. This layer is not required to be ported when the underlying hardware changes.

CMakeLists.txt

contains information about CMake build support.

README.md

README file for PSA test suite.

2.6 Feedback and contributions

For feedback, use the GitHub Issue Tracker that is associated with this repository.

Arm licensees can contact Arm directly through their partner managers.

Arm also welcomes code contributions through GitHub pull requests. See the GitHub documentation on how to raise pull requests.

3. Validation methodology

This chapter describes the validation methodology used for the PSA functional API test suite.

3.1 Test layering details

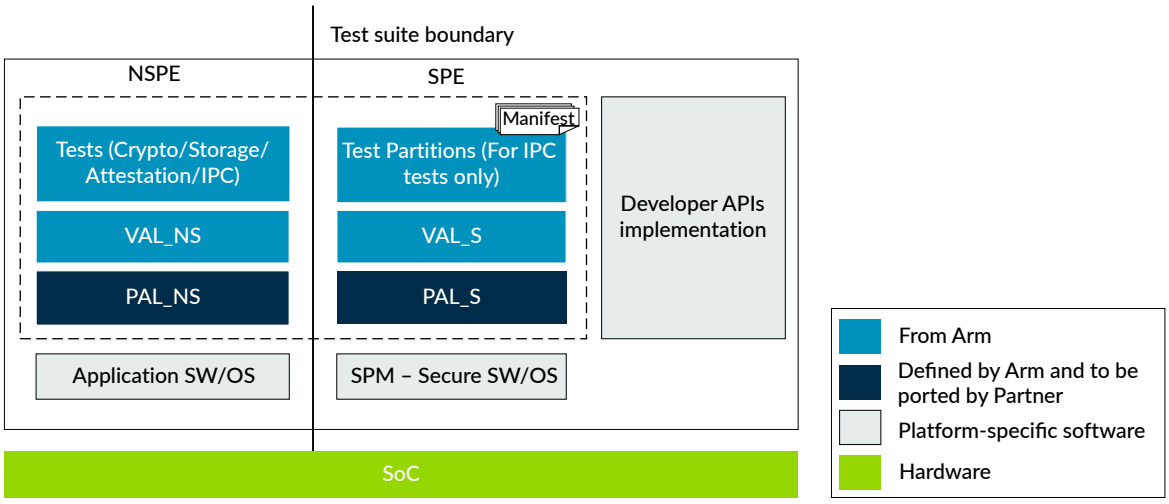
PSA tests are self-checking and portable C-based tests with directed stimulus. These tests use the layered software stack approach to enable porting across different test platforms.

The constituents of the layered stack are:

- Tests
- Secure partitions
- VAL
- PAL

The following figure illustrates the layered software stack approach.

Figure 3-1: Layered software stack



The following table describes the constituents of the layered stack.

Table 3-1: Layered software stack components

Layer	Description
Tests	A set of C-based directed tests, each of which verifies the implementation against a test scenario that is described by the PSA specification. These tests include checks related to PSA-FF and functional APIs, and are expected to be run in Non-secure mode. PSA-FF tests may further use IPC calls to communicate test suite-defined Secure partition to cover the appropriate test scenario. These tests are abstracted from the underlying hardware platform by the VAL. This implies that porting a test for a specific target platform is not required.
Secure partitions	PSA-FF test suite defines three Secure partitions: • Driver partition provides driver-related services such as print API to the PSA test suite Non-secure code and to the other partitions. • Client partition drives the Secure client test functions for the IPC tests. • Server partition drives the Secure server test functions for the IPC tests. These Secure partitions must be integrated into your Secure software containing SPM. They are valid only for IPC tests. Functional APIs tests are not required to use these partitions. Secure partition-related manifest files are available in the `platform/manifests/` directory.
VAL	This layer provides a uniform and consistent view of the available test infrastructure to the tests in the test pool by making appropriate calls to the PAL. It is designed such that it can be used both from Secure and Non-secure sides. This layer does not require porting when the underlying hardware changes.
PAL	This layer is the closest to the hardware and is aware of the platform details. It is responsible for presenting the hardware through a consistent interface to VAL. This layer must be ported to the specific hardware present in the platform. The PAL is designed such that it can be used from both Secure and Non-secure sides.

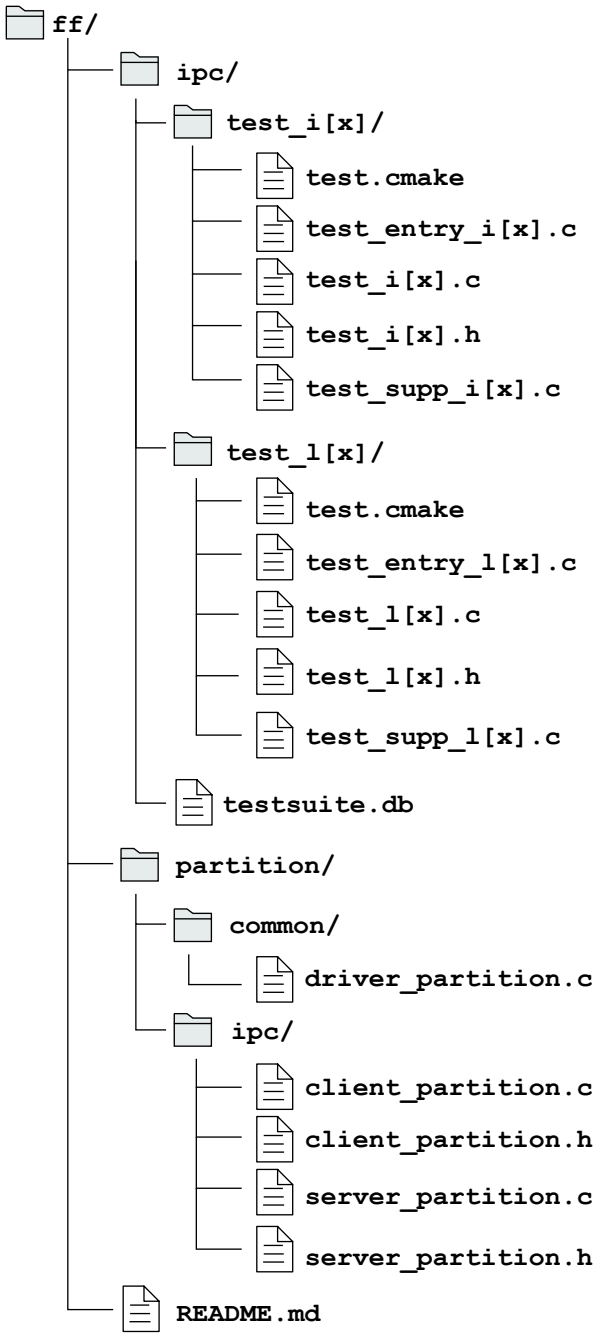
3.2 Test suite organization

The directory structures of PSA-FF and functional APIs test suites are described in this section.

PSA-FF test suite

The following figure shows the contents of the directories, subdirectories, and files in the PSA-FF test suite.

Figure 3-2: PSA-FF test suite



ipc Holds IPC tests.

test_[y][x] Test directory containing IPC test related files. Here, `y` is:

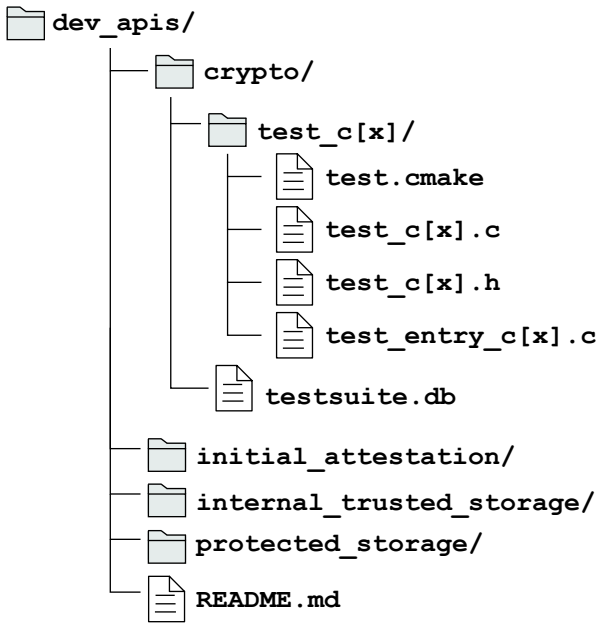
- `i` for IPC tests.
- `l` for lifecycle tests.

test.cmake	Helps to identify the test files that must be compiled to generate the test binaries.
test_entry_i[x].c	Holds the test entry point in NSPE and executes test functions from NSPE. For IPC tests, it can execute the same test functions from SPE, based on the test requirement.
test_[y][x].c and **test_[y][x].h**	`test_[y][x].c` and `test_[y][x].h`.
test_supp_[y][x].c	Holds server test functions.
testsuite.db	A database file representing tests to be compiled and run as part of specific suite. This provides flexibility to run specific tests individually by commenting out the other tests.
partition	Contains partition files that provide different driver services to the tests and the dispatcher logic to dispatch specific client or server test functions.
README.md	This file contains information for building the PSA-FF test suite.

Functional APIs test suite

The following figure shows the contents of the directories, subdirectories, and files in the functional APIs test suite.

Figure 3-3: Functional APIs test suite



crypto	Holds Crypto tests.
test_[x][y]	Test directory containing test-related files. `[x]` can be: • `c` for Crypto tests

- `a` for Initial Attestation
- `p` for Protected Storage
- `s` for Internal Trusted Storage

`[y]` is the test number.

`test.cmake`	Helps to identify the test files that must be compiled to generate the test binaries.
`test_[x][y].c` and **`test_[x][y].h`**	Hold the actual test functions.
`test_entry_c[x].c`	Holds the test entry point in NSPE and executes test functions from NSPE.
`testsuite.db`	A database file representing tests to be compiled and run as part of specific suite. This provides flexibility to run specific tests individually by commenting out the other tests.
`initial_attestation`	Holds Initial Attestation tests.
`internal_trusted_storage`	Holds Internal Trusted Storage tests.
`protected_storage`	Holds Protected Storage tests.
`README.md`	This file contains information for building the functional APIs test suite.

3.3 Test execution flow

This section provides details of the test execution flows for PSA-FF tests and functional APIs tests.

PSA-FF tests

The test compilation tool generates the NSPE and SPE archives for IPC tests. You must integrate test suite SPE archives with your Secure software stack containing the SPM, such that it gets access to PSA-defined client APIs and Secure partition APIs. The NSPE libraries generated by the test suite must be integrated with the NSPE OS such that test suite NSPE code gets access to the PSA-defined client APIs.

For more information on IPC test archives, see 3.4 Integrating the test suite with the SUT on page 21.

The System Under Test (SUT) boots to an environment that enables the test functionality. This implies that the SPM is initialized, and PSA-FF partitions are ready to accept requests.

On the Non-secure side, the SUT boot software gives control to the tests entry point (`val_entry` symbol) as an application entry point in Non-secure privileged mode.

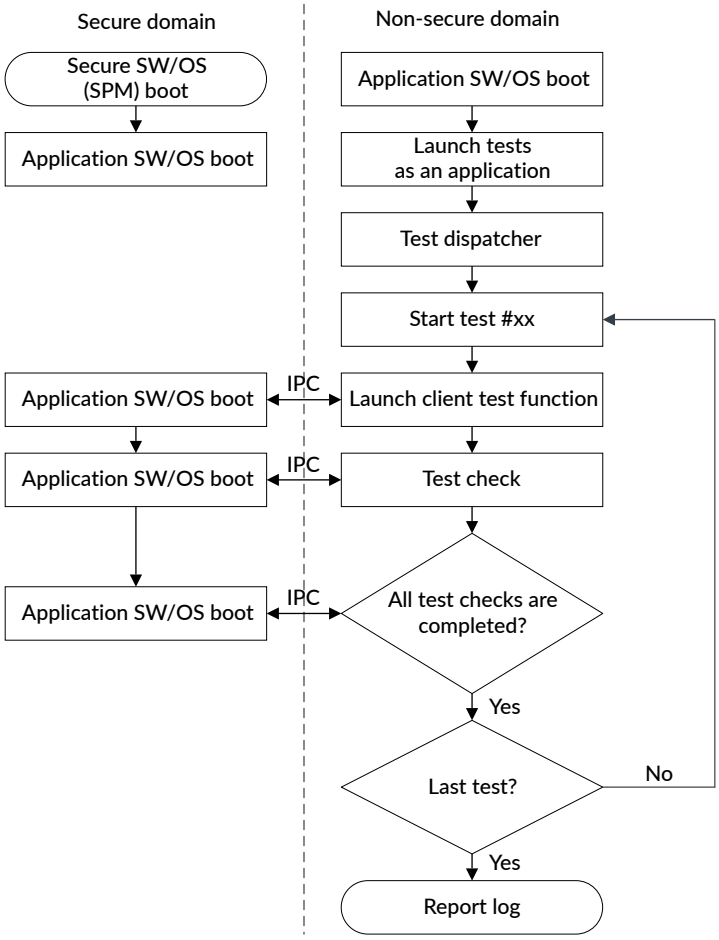
The PSA tests query the VAL layer to get the necessary information to run the tests. This information can include memory maps, interrupt maps, and hardware controller maps.

Based on the test scenario, the test and partition communicate with each other using IPC APIs that are defined in the specification, and report the test results using VAL print API (in turn PAL API ported to the specific platform). Each IPC test scenario is driven using dedicated client-server tests functions. The client functions are available in `test_ix.c` and are suffixed with `client_test_` label. Based on test requirements, client functions are executed either in NSPE or SPE or both. Server functions are available in `test_supp_ix.c` and are suffixed with `server_test` label. They are always executed in SPE.

All the tests are executed sequentially. The dispatcher in the VAL queries the next test on the completion of the present test. The dispatcher also makes VAL (and in turn PAL) calls to save and reports each of the test results.

For more information on the dispatcher, see 3.5 Test dispatcher on page 23.

Figure 3-4: Test execution flow for PSA-FF IPC tests



Functional APIs tests

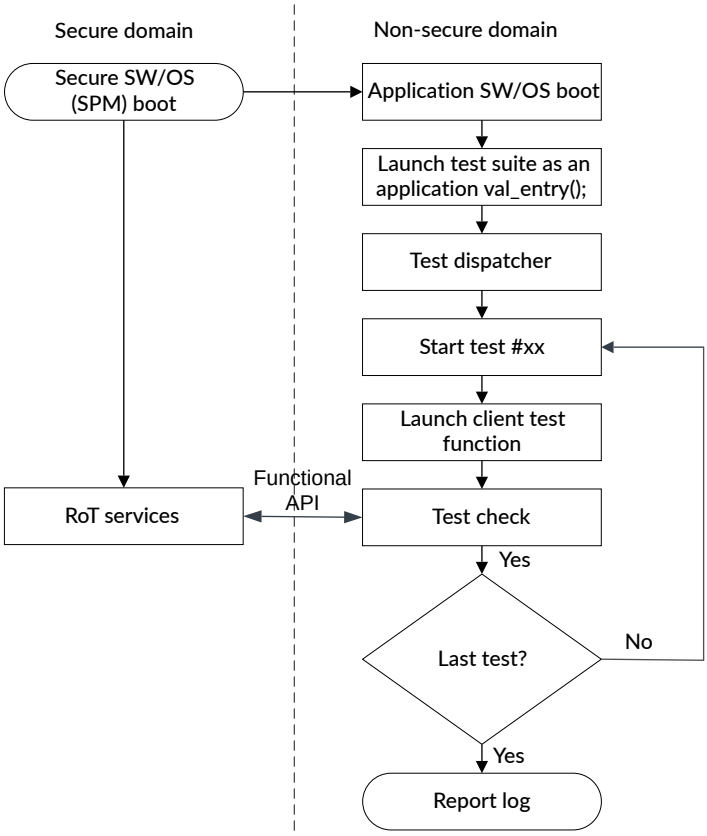
You must integrate the test suite NSPE archives with your Non-secure software stack such that it gets access to PSA-defined functional APIs. The SUT then boots to an environment that enables the test functionality. The SUT boot software gives control to the test entry point (`val_entry` symbol) as an application entry point in the Non-secure privileged mode.

The test compilation tool generates the NSPE archives for functional tests as described in 3.4 Integrating the test suite with the SUT on page 21.

The tests query the VAL to get necessary information to run the tests. This information can include memory maps, interrupt maps, and hardware controller maps. Based on the test scenario, the test calls functional APIs and reports the test results using the VAL print API (in turn PAL API ported to the specific platform).

All the tests are executed sequentially. The dispatcher in the VAL queries the next test on the completion of the present test. For more information on the dispatcher, see 3.5 Test dispatcher on page 23.

Figure 3-5: Test execution flow for functional APIs tests



3.4 Integrating the test suite with the SUT

The test compilation flow creates the following libraries that you must integrate with your SUT software.

- **Test framework**

  The test compilation flow creates two archive files that contain code for the test framework (VAL and PAL APIs), and the test dispatcher logic that must be available in the

main memory and executed as an application in NSPE. Link these archives with the NS OS library to generate an NSPE binary.

- `<BUILD_PATH>/BUILD/val/val_nspe.a`
- `<BUILD_PATH>/BUILD/platform/pal_nspe.a`

- **Combined tests archive**

  The test compilation flow generates a combined test archive by combining all the Non-secure test objects for Non-secure tests. The generated archive is placed at `<BUILD_PATH>/<top_level_suite>/<suite>/test_combine.a`. Integrate this archive library with the test framework libraries and NS OS library to generate an NSPE binary. The dispatcher function within the VAL calls each test entry function one after another, to run the Non-secure tests.

- **Test suite Secure partitions**

  Along with test framework and combined tests libraries, the IPC tests require the SPE binaries. The test suite compilation flow generates the following Secure partition archives for IPC tests. You must integrate these test suite partition archives with your SPE code such that it follows the level of isolation rules defined in the PSA-FF specification. Load the resultant SPE binary into the Secure main memory.

Table 3-2: Libraries and protection domains

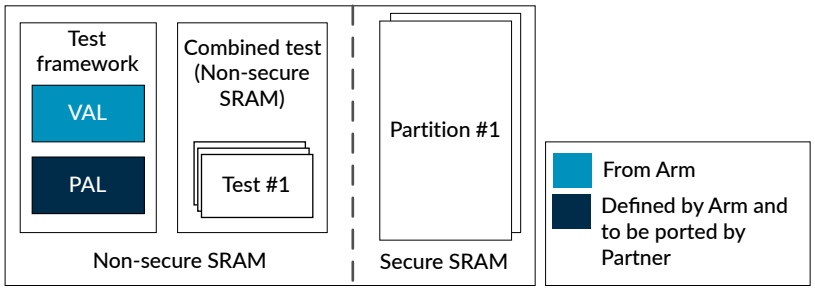
Test suite partition libraries	Protection domain
`<build_dir>/BUILD/partition/driver_partition.a`	PSA-RoT
`<build_dir>/BUILD/partition/client_partition.a`	Application-RoT
`<build_dir>/BUILD/partition/server_partition.a`	Application-RoT

Note

- The client and server test functions of all the tests are compiled as part of `client_partition` and `server_partition` respectively. All these functions are loaded into the Secure main memory and are available at same time.
- If an SUT has main memory size constraints, you can compile and run the tests in a bulk of test sets, for example, 10 tests at time. To do this, remove the test references other than the ones required from the respective suite specific `testsuite.db` file. Repeat this process for all the test sets.

The following figure shows the loading test binaries.

Figure 3-6: Loading test binaries



3.5 Test dispatcher

The dispatcher has certain responsibilities. Each test must present the `test_entry` function address to the dispatcher. To this function, the dispatcher passes a pointer to a structure containing the function pointers to all the available VAL functions. These functions make the appropriate VAL function call.

The flow of the dispatcher is as follows:

1. Query the test_entry function address.
2. Call the test_entry function of the test and execute the tests.
3. Wait for completion of the test.
4. Print and save the result of the test.
5. Repeat steps 1-4 until the end of the last test.
6. Report the test suite result summary.

To facilitate test reporting and management of observing aspects, the PSA-FF system contains UART for printing the status of tests. If a display console is not available, PAL can be updated to make the test results available to the external world through other means.

Information about the environment in which a host test harness is running, is beyond the scope of this document. However, it is presumed that the SUT is communicating with the host using serial port, JTAG, Wi-Fi, USB, or any other means that allow for access to the external world.

3.6 Analyzing test run results

Each test follows a uniform test structure that is defined by VAL.

- Performing any test initializations.
- Dispatching the test functions.
- Waiting for test completion.
- Performing the test exit.

The test may pass, fail, skip, or be in an error state. For example, if the test times out or the system hangs, it means that something went wrong and the test framework was unable to determine what happened. In this case, you may have to check the logs. If a test fails or skips, you may see extra print messages to determine the cause.

The test suite summary is displayed at the end. An example of the test suite summary is shown below.

```
***** PSA Architecture Test Suite - Version x.y *****
```

```
Running.. Crypto Suite
*********************************************

TEST: 201 | DESCRIPTION: Testing psa_crypto_init API: Basic
TEST RESULT: PASSED

*********************************************

TEST: 202 | DESCRIPTION: Testing crypto key management APIs
Failed at Checkpoint : 3
Actual               : 1
Expected             : 0
TEST RESULT          : FAILED (Error Code=0x1)

*********************************************

************** Crypto Suite Report *************
TOTAL TESTS          : 2
TOTAL PASSED         : 1
TOTAL SIM ERROR      : 0
TOTAL FAILED         : 1
TOTAL SKIPPED        : 0
*********************************************

Entering standby..
```

Debugging of a failed test

Each test is organized with a logical set of self-checking code. If a failure occurs, searching for the relevant self-checking point is a useful point to start debugging.

Consider the above snippet of a failing test on the display console.

Here are some debugging points to consider.

- If the default prints do not give enough information, you can recompile and rerun the test binaries with high print verbosity level. See the PSA test suite build README to understand how test verbosity can be changed.
- In the above example, test 2 is failing. This test is located at `dev_apis/crypto/test_c002/`
- Since the failure message is shown as checkpoint 3, go to this print point in the test source code and debug the failing cause. The checkpoints are reserved in the test suite as shown below:
  - Checkpoints 1-100 are reserved for functional APIs tests. Checkpoints print messages with numbers which can come from `test_[x][y].c` file. Here, `[x]` is reserved for functional API tests and `[y]` is the test number.
  - Checkpoints 101-200 are reserved for client test functions of IPC tests and prints related to these numbers can come from `test_i[y].c`
  - Checkpoints 201-300 are reserved for server test functions of IPC tests and prints related to these numbers can come from `test_supp_i[y].c`
- Status of the failure code (0x1 in this example) is mapped with a structure `val_status_t` that is available at `val/common/val.h`. Look for enum that is dedicated to this number to see the status in verbatim form.

Appendix A Revisions

This appendix describes the technical changes between released issues of this book.

A.1 Revisions

This section provides details of all the technical changes between different versions of this book.

Table A-1: Issue A

Change	Location
This is the first revision of the document.	-

Table A-2: Differences between Issue A and Issue B

Change	Location
Updated the path to secure manifest files.	See 3.1 Test layering details on page 15
Updated the test execution flow and SPE binary information.	See the following sections: • 3.3 Test execution flow on page 19 • 3.4 Integrating the test suite with the SUT on page 21

Table A-3: Differences between Issue B and Issue C

Change	Location
Added information about Functional APIs.	See the following sections: • 2.2 PSA APIs on page 10 • 2.3 Test suite on page 12 • 2.5 Directory structure on page 13 • 3.2 Test suite organization on page 16 • 3.3 Test execution flow on page 19
Added ITS and PS information.	See the following sections: • 2.1 Abbreviations on page 10 • 2.3 Test suite on page 12
Moved information about the test dispatcher to a new section.	See 3.5 Test dispatcher on page 23
Updated the test suite summary and debugging details.	See 3.6 Analyzing test run results on page 23

Table A-4: Differences between Issue C and Issue D

Change	Location
Added PSA RoT sub category.	See 2.3 Test suite on page 12.
Updated details about the compliance sign-off process.	See Compliance sign-off process.
Added lifecycle test directory in the PSA-FF directory structure.	See 3.2 Test suite organization on page 16.
Updated the section with details about integrating the test suite with the SUT.	See 3.4 Integrating the test suite with the SUT on page 21.

Table A-5: Differences between Issue D and Issue E

Change	Location
Added `CMakeLists.txt` to the directory structure.	See 2.5 Directory structure on page 13.
Updated `source.mk` and `test_entry.c` to `test.cmake` and `test_entry_i[x].c` respectively.	See 3.2 Test suite organization on page 16.
Updated the information about PSA-FF and Functional APIs test execution.	See 3.3 Test execution flow on page 19.
• Updated the combined test archive section. • Updated the image for loading test binaries.	See 3.4 Integrating the test suite with the SUT on page 21.
Updated the dispatcher flow.	See 3.5 Test dispatcher on page 23.

Table A-6: Differences between Issue E and Issue F

Change	Location
Removed the compliance sign-off process section from Introduction.	See 2. Introduction to PSA test suite on page 10.
Updated the description for Secure partitions.	See 3.1 Test layering details on page 15.

Table A-7: Differences between Issue F and Issue G

Change	Location
No technical changes.	-

Table A-8: Differences between Issue G and Issue 0102-01

Change	Location
No technical changes.	-

Table A-9: Differences between Issue 0102-01 and Issue 0103-01

Change	Location
No technical changes.	-

Table A-10: Differences between Issue 0103-01 and Issue 0104-01

Change	Location
No technical changes.	-

Table A-11: Differences between Issue 0104-01 and Issue 0105-01

Change	Location
Updated PSA Architecture Test suit version to 1.5 in Analyzing test run results.	See 3.6 Analyzing test run results on page 23.

Table A-12: Differences between Issue 0105-01 and Issue 0106-01

Change	Location
Updated the PSA Architecture Test Suite Version to x.y.	See 3.6 Analyzing test run results on page 23.
Added PSA certified secure storage and PSA attestation in Arm product resources.	See [PSA-M Functional API test suite] Additional Reading Template.