Arm® Platform Security Architecture APIs Test Suite

Revision: r0p0

Validation Methodology

Arm® Platform Security Architecture APIs 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.

Contents

Arm® Platform Security Architecture APIs Test Suite Validation Methodology

Preface

This preface introduces the *Arm® Platform Security Architecture APIs Test Suite Validation Methodology*.

It contains the following:

About this book

This book describes the test suite for Platform Security Architecture APIs.

Product revision status

The r*m*p*n* identifier indicates the revision status of the product described in this book, for example, r1p2, where:

r*m* Identifies the major revision of the product, for example, r1.

p*n* Identifies the minor revision or modification status of the product, for example, p2.

Intended audience

This book is written for engineers who are specifying, designing, or verifying an implementation of the Arm® Platform Security Architecture Firmware Framework architecture.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction
This chapter introduces the features and components of the test suite for Arm Platform Security Architecture APIs.

Chapter 2 Validation methodology
This chapter describes the validation methodology that is used for the test suite.

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

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.

Typographic conventions

italic
Introduces special terminology, denotes cross-references, and citations.

bold
Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`monospace` (underlined: mono)
Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
Denotes language keywords when used outside example code.

`<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
: Used in body text for a few terms that have specific technical meanings, that are defined in the *Arm® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Timing diagrams

The following figure explains the components used in timing diagrams. Variations, when they occur, have clear labels. You must not assume any timing information that is not explicit in the diagrams.

Shaded bus and signal areas are undefined, so the bus or signal can assume any value within the shaded area at that time. The actual level is unimportant and does not affect normal operation.

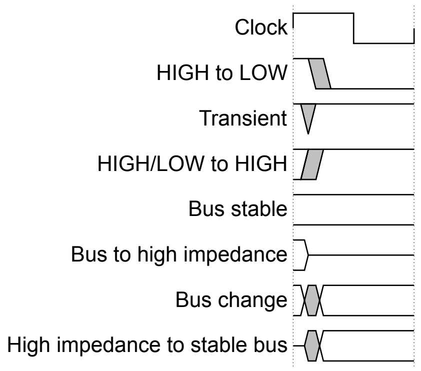


Figure 1 Key to timing diagram conventions

Signals

The signal conventions are:

Signal level

The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals.
- LOW for active-LOW signals.

Lowercase n

At the start or end of a signal name denotes an active-LOW signal.

Additional reading

This book contains information that is specific to this product. See the following documents for other relevant information.

Arm publications

- *Arm® Platform Security Architecture Firmware Framework specification* (ARM DEN 0063).
- *PSA Security model* (ARM DEN 0079).
- *Arm® Trusted Base System Architecture for Armv8-M* (ARM DEN 0062A).
- *PSA Trusted Boot and Firmware Update* (ARM DEN 0072A).
- *PSA Crypto API*
- *Armv8 Architecture Reference Manual, Armv8 for M-profile* (Arm DDI 00553A)

Other publications

None.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Arm Platform Security Architecture APIs Test Suite Validation Methodology*.
- The number 101447_0000_C_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

Note

Arm tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Chapter 1
Introduction

This chapter introduces the features and components of the test suite for Arm Platform Security Architecture APIs.

It contains the following sections:

1.1 Scope of the document

The goal of this document is to describe the validation methodology for PSA APIs test suites. It focuses on describing the framework and the methodology within which the tests are run.

1.2 Abbreviations

This section lists the abbreviations that are used in this document.

Table 1-1 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
SID	Secure function IDentifier
SPE	Secure Processing Element
SPM	Secure Partition Manager
SUT	System Under Test
VAL	Validation Abstraction Layer

1.3 Platform Security Architecture APIs

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

PSA provides a recipe, based on industry best practice, that allows security to be consistently designed in, at both a hardware and firmware level. 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 sets of API specifications:
- PSA Firmware Framework
- PSA developer APIs

This section contains the following subsections:
- *1.3.1 PSA Firmware Framework* on page 1-13.
- *1.3.2 PSA developer APIs* on page 1-14.

1.3.1 PSA Firmware Framework

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

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 typically 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 or 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 might 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 the fundamental RoT Services to the system and also manages the isolated execution environment for the Application RoT Services.

The main components of PSA RoT are described in the following table.

Table 1-2 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	Provide essential cryptographic functionality and manage accesses to the immutable RoTs for Application RoT services.

The Firmware Framework specification:

- Provides requirements for the 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 details on SPM and PSA RoT, refer to the specification documents mentioned in the *Additional reading* section of this document.

1.3.2 PSA developer APIs

PSA developer APIs are the top-level APIs used by application developers and RTOS vendors. These APIs have been designed for software developers who want to implement hardware security features without necessarily being security experts themselves.

These APIs provide the top-level essential services related to crypto, secure storage, and attestation tokens. For details, see the *Developer APIs specification*.

1.4 Test suite

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

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

Table 1-3 Test categories and their descriptions

API type	Test category	Sub category	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 verifying the correctness of client APIs.
		Secure partition APIs	Tests verifying the correctness of Secure partition APIs.
		Manifest input	Tests verifying manifest input parameters.
Developer APIs	Crypto	PSA crypto APIs	Tests verifying the correctness of PSA crypto APIs.
	Internal Trusted Storage (ITS)	PSA ITS APIs	Tests verifying the correctness of PSA ITS APIs.
	Protected Storage (PS)	PSA PS APIs	Tests verifying the correctness of PSA PS APIs.
	Initial Attestation	PSA Initial Attestation APIs	Tests verifying the correctness of PSA Initial Attestation APIs.

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 for compliance, see test-list documents in the `doc/` directory.

1.5 Test suite components

The components of the test suite are described in the following table:

Table 1-4 Test suite components

Component	Description
Test suites	Contain self-checking tests that are written in C.
Substructure	Test supporting layers consist of a framework and libraries setup as: • Tools to build the compliance tests • VAL library • PAL library
Documentation	Suite-specific documents such as testlists, porting guide, and API specification.

1.6 Directory structure

Components of the tests must be in a specific hierarchy for the test suite. When the release package is downloaded from GitHub, the top-level directory contains the files that are shown in the following figure.

```
api-tests/
 │
 ├──dev_apis
 │
 ├──docs
 │
 ├──ff
 │
 ├──platform
 │
 ├──tools
 │
 ├──val
 │
 └──README.md
```

Figure 1-1 Test suite directory structure

dev_apis

This directory contains subsuites containing architecture tests for the developer 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 developer APIs specification. These tests are abstracted from the underlying hardware platform by the VAL.

docs

This directory contains the test suite documentation.

ff

This directory contains 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

This directory contains files to form 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

This directory contains make files and scripts that are used to generate test binaries.

val

This directory 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. The VAL makes appropriate calls to the PAL to achieve this functionality. This layer is not required to be ported when the underlying hardware changes.

README.md

README file for PSA test suite.

1.7 Compliance sign-off process

The future releases of this specification will be updated with compliance sign-off process and expectations from partner on the process for running PSA APIs test suite.

1.8 Feedback, contributions, and support

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

For support, send an email to *support-psa-arch-tests@arm.com* with the details.

Arm licensees can contact Arm directly through their partner managers.

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

Chapter 2
Validation methodology

This chapter describes the validation methodology that is used for the test suite.

It contains the following sections:

2.1 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

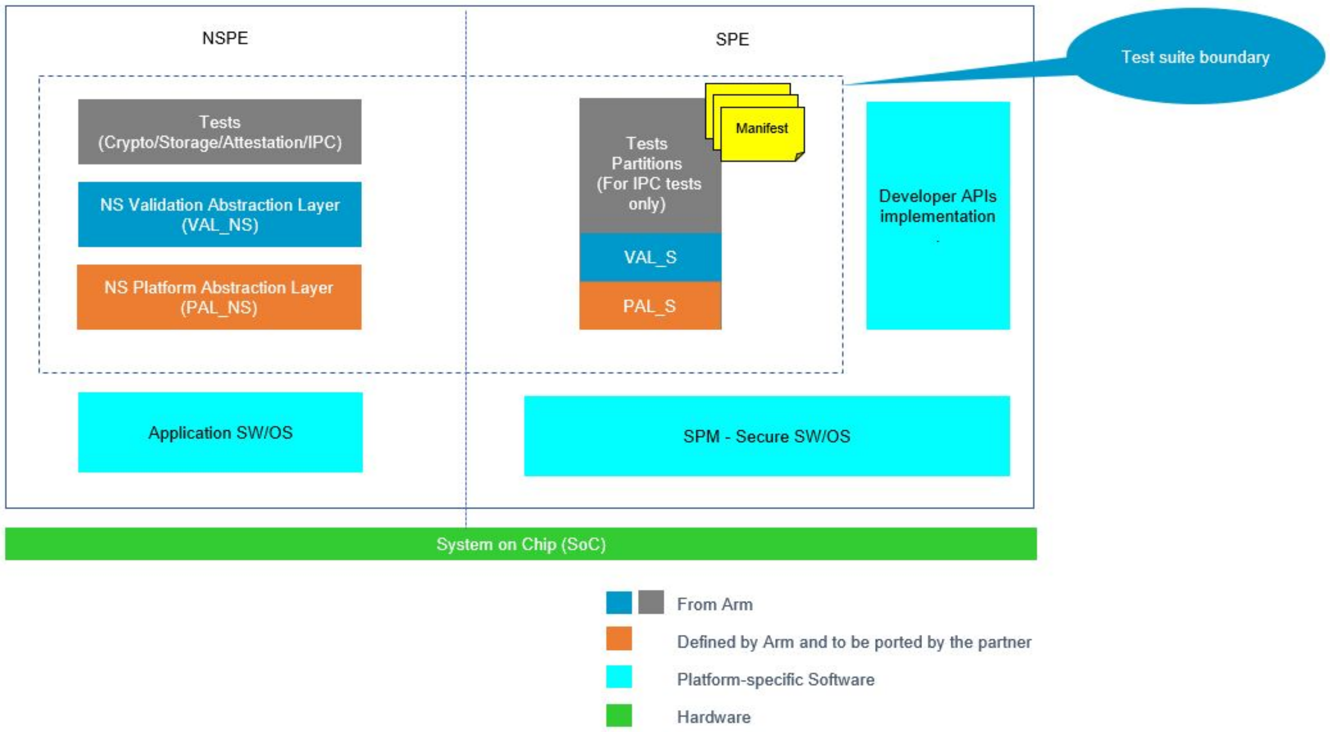

NSPE
SPE
Test suite boundary
Tests (Crypto/Storage/Attestation/IPC)
NS Validation Abstraction Layer (VAL_NS)
NS Platform Abstraction Layer (PAL_NS)
Tests Partitions (For IPC tests only)
Manifest
VAL_S
PAL_S
Developer APIs implementation
Application SW/OS
SPM - Secure SW/OS
System on Chip (SoC)
From Arm
Defined by Arm and to be ported by the partner
Platform-specific Software
Hardware

Figure 2-1 Layered software stack

The following table describes the constituents of the layered stack.

Table 2-1 Layered software stack components

Layer	Description
Tests	A set of C-based directed tests, each of which verifies the implementation against a test scenario described by the PSA specification. These tests include checks related to PSA-FF and developer APIs, and are expected to be run in Non-secure. 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 it is not required to port a test for a specific target platform.
Secure partitions	PSA-FF IPC tests contain the secure partitions that define secure functions required for covering IPC test scenarios. These secure partitions must be integrated into your secure software containing SPM. These secure partitions are valid only for IPC tests. Developer APIs tests are not required to use these partitions. The secure partition related manifest files are available at: • `platform/targets/<targetName>/manifests/common/` directory. Partitions formed using this manifest provide driver related services such as UART print. • `platform/targets/<targetName>/manifests/ipc/` directory. This directory contains the manifests for client and server partitions. Secure functions defined in these manifests are used by IPC tests to cover appropriate test scenarios.
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. The VAL is designed such that it can be used both from Secure and Non-secure sides. This layer is not required to be ported 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.

2.2 Test suite organization

The directory structures of PSA-FF and developer APIs test suites are discussed 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.

```
ff
│
├──ipc
│   ├──test_i[x]
│   │       ├───────source.mk
│   │       ├───────test_entry.c
│   │       ├───────test_i[x].c
│   │       ├───────test_i[x].h
│   │       └───────test_supp_i[x].c
│   └──testsuite.db
├──partition
│   ├──common
│   │       └───────driver_partition.c
│   └──ipc
│          ├────────client_partition.c
│          ├────────client_partition.h
│          ├────────server_partition.c
│          └────────server_partition.h
└───README.md
```

Figure 2-2 PSA-FF test suite directory structure

Table 2-2 Directory content

Directory	Content
ipc	Holds IPC tests.
test_i[x]	Test directory containing IPC test related files.
source.mk	Helps to identify the test files that must be compiled to generate the test binaries.
test_entry.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_i[x].c and test_i[x].h	Holds the client test functions. [x] is the test number.
test_supp_i[x].c	Holds server test functions. This is available only for IPC tests.
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 the other tests out.
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.

Developer APIs test suite

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

```
dev_apis
├──crypto
│    ├──test_c[x]
│    │    ├──────source.mk
│    │    ├──────test_c[x].c
│    │    ├──────test_c[x].h
│    │    └──────test_entry.c
│    └──testsuite.db
├──initial_attestation
├──internal_trusted_storage
├──protected_storage
└──README.md
```

Figure 2-3 Developer APIs test suite directory structure

Table 2-3 Developer APIs directory contents

Directory or file	Content
`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.
`source.mk`	Helps to identify the test files that must be compiled to generate the test binaries.
`test_[x][y].c` and `test_[x][y].h`	Holds the actual test functions.
`test_entry.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 the other tests out.
`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 developer APIs test suite.

2.3 Test execution flow

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

PSA-FF tests

The test compilation tool generates the NPSE and SPE archives for IPC tests. For details about IPC test archives, see *2.4 Loading test suite binaries* on page 2-28. You must integrate test suite SPE archives with your Secure software stack contaning 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.

Then 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 needs, 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.

Due to RAM and flash size constraints, all the tests may not be available at the same time. The dispatcher in the VAL queries the PAL to load the next test on the completion of the present test. The PAL may optionally communicate with the external world to load the next test. The dispatcher also makes VAL (and in turn PAL) calls to save and reports each of the test results. For details about the dispatcher, see *2.5 Test dispatcher* on page 2-30.

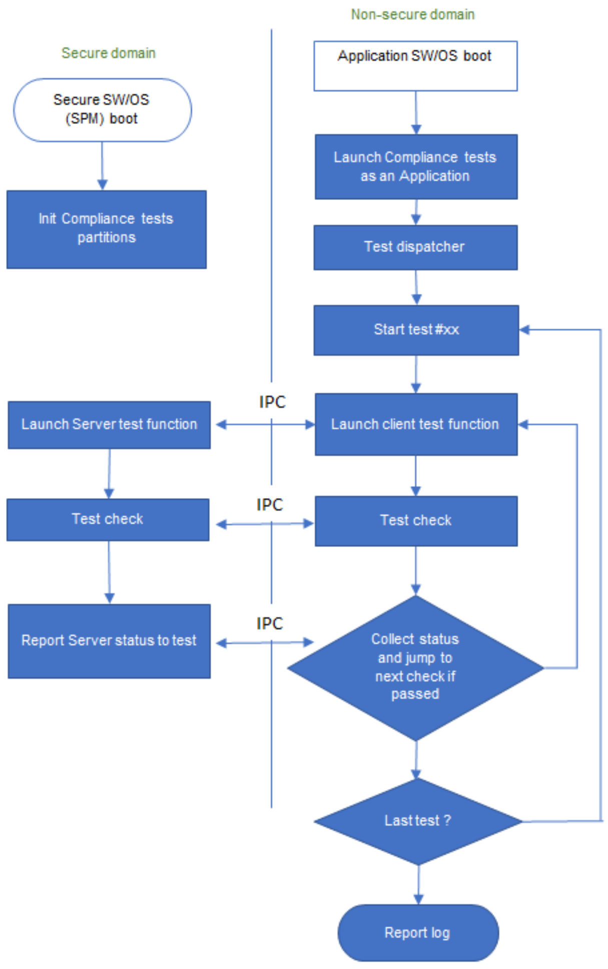

Secure domain
Non-secure domain
Secure SW/OS (SPM) boot
Init Compliance tests partitions
Application SW/OS boot
Launch Compliance tests as an Application
Test dispatcher
Start test #xx
IPC
Launch Server test function
Launch client test function
IPC
Test check
Test check
IPC
Report Server status to test
Collect status and jump to next check if passed
Last test ?
Report log

Figure 2-4 Test execution flow for PSA-FF tests

Developer APIs tests

The test compilation tool generates the NPSE archives for developer tests as decribed in the *2.4 Loading test suite binaries* on page 2-28 section.

You must integrate the test suite NSPE archives with your Non-secure software stack such that it gets access to PSA defined developer 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 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 developer APIs and reports the test results using the VAL print API (in turn PAL API ported to the specific platform).

Due to RAM and flash size constraints, all the tests may not be available at the same time. The dispatcher in the VAL queries the PAL to load the next test on the completion of the present test. The PAL may optionally communicate with the external world to load the next test. The dispatcher also makes VAL, and in turn PAL calls to save and reports each of the test results. For information about the dispatcher, see *2.5 Test dispatcher* on page 2-30.

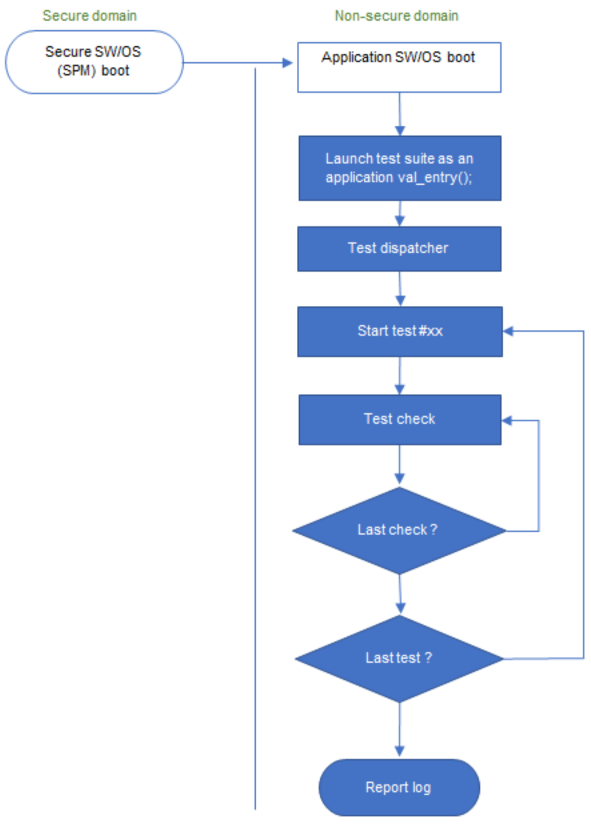


Figure 2-5 Test execution flow for developer APIs tests

2.4 Loading test suite binaries

An SUT can have main memory (SRAM or Flash) size constraints. It is possible that all the tests do not fit into these types of memories and may not be available at the same time. If an SUT does not have this limitation, test binaries can be stored in main memory directly. Otherwise, they can be stored in the secondary memory.

The following binaries are required to be loaded into memory:

- NSPE binary
  Test compilation flow creates two archive files that contain code for the test framework - VAL and PAL APIs, and test dispatcher logic that must be available in the main memory and executed as an application in NSPE. Link these archives with NS OS library to be able to generate an NSPE binary.
  — `<BUILD_PATH>/BUILD/val/val_nspe.a`
  — `<BUILD_PATH>/BUILD/platform/pal_nspe.a`
- Combined test binary

  Test compilation flow also creates a combined test binary containing all Non-secure test binaries together at `<BUILD_PATH>/BUILD/<top_level_suite>/<suite>/test_elf_combine.bin`. You can load this binary into Non-secure main memory if the SUT has enough space. Otherwise, this can be loaded into secondary storage. The dispatcher function within the VAL reads this binary and loads each of test sections using PAL API to Non-secure memory one after another. The addresses of the various sections must be provided using `target.cfg`.
- Along with NSPE binaries and combined test binary, IPC tests require the following SPE binaries.

  Test suite compilation flow generates the following Secure partition archives for IPC tests. You must integrate these test suite partitions archives with your SPE code and load the resultant SPE binary into Secure main memory. Note that the client test functions and server test functions of all 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.

  Test suite partition libraries:
  — `<build_dir>/BUILD/partition/driver_partition.a`
  — `<build_dir>/BUILD/partition/client_partition.a`
  — `<build_dir>/BUILD/partition/server_partition.a`

——— **Note** ———

If an SUT has main memory size constraints, you can compile and run these test functions in a bulk of test sets, for example, 10 tests at time. To do this, remove test references other than the required test bulk test set from the respective suite specific `testsuite.db` file. Repeat this process for all the test sets.

———————————

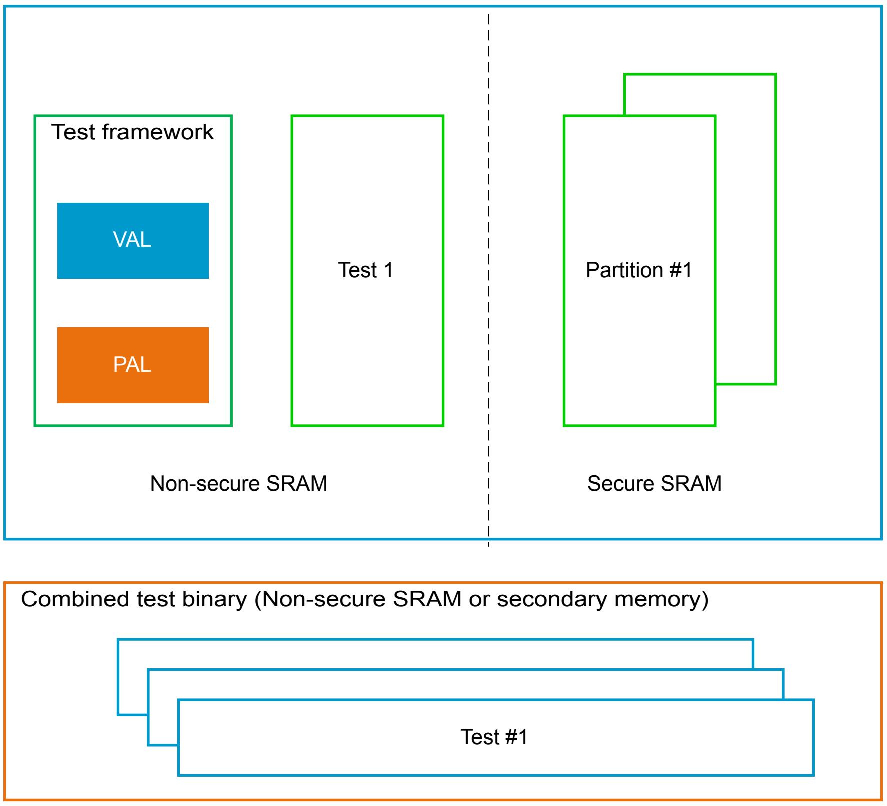


Figure 2-6 Loading test binaries

2.5 Test dispatcher

The dispatcher has the following 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. Request VAL to load the metadata of the next test into the main memory.
2. Parse the metadata that is associated with the test.
3. Verify that the test is compatible with the SUT.
4. Load the test code and data sections into the appropriate locations in the main memory.
5. Call the `test_entry` function of the test and execute tests.
6. Wait for test completion.
7. Print and save the result of the test.
8. Repeat steps 1-6 till the end of the last test.

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, the 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.

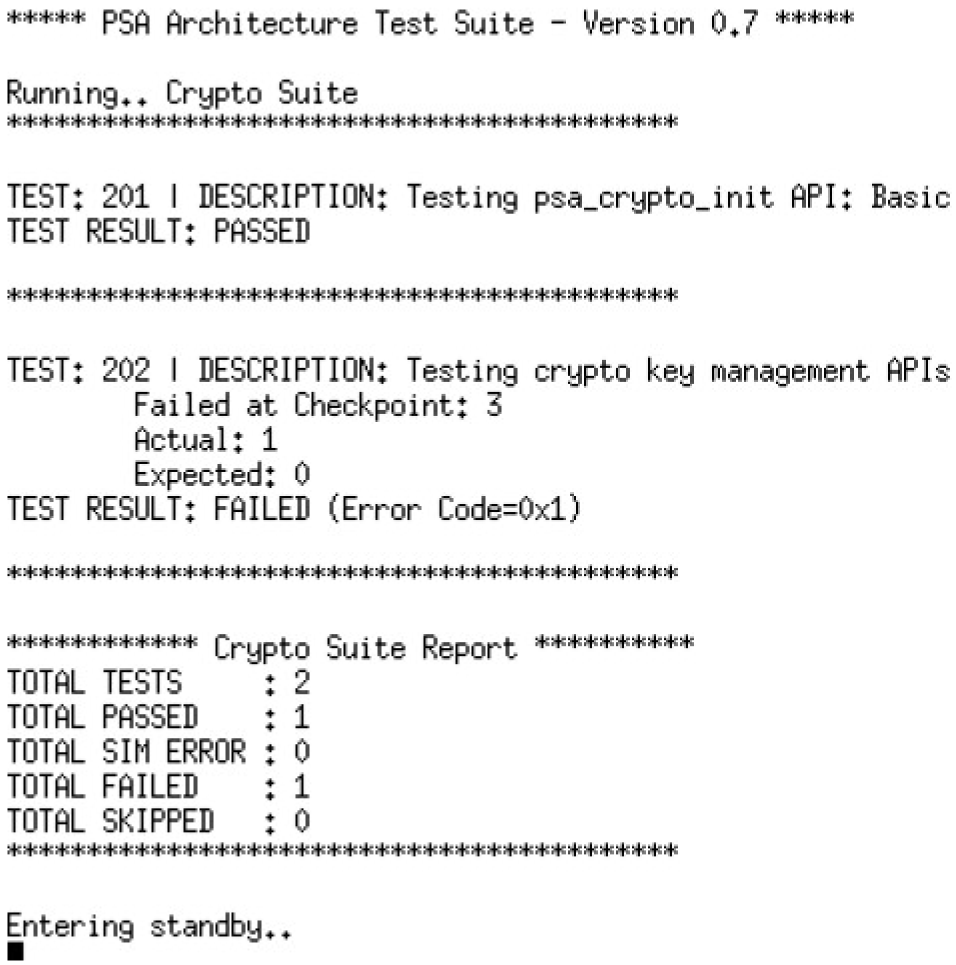
2.6 Analyzing test run results

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

1. Performing any test initializations.
2. Dispatching the client-server test functions.
3. Waiting for test completion.
4. Performing the test exit.

The test may pass, fail, skip or be in an error state. For example, if 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 snapshot of the test suite summary is shown in the following figure.



```
***** PSA Architecture Test Suite - Version 0.7 *****

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..
█
```

Figure 2-7 Test suite summary

Debugging of a failing test

Since each test is organized with a logical set of self-checking code, in the event of a failure, 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 `--verbose` switch in the `setup.sh` script 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 developer APIs tests. Checkpoints print messages with numbers which can come from `test_[x][y].c` file. Here, `[x]` is reserved for developer 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 dedicated to this number to see status in verbatim form.

Appendix A
Revisions

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

It contains the following section:

A.1 Revisions

Table A-1 Issue A

Change	Location	Affects
This is the first revision of the document.	-	All revisions

Table A-2 Differences between Issue A and Issue B

Change	Location	Affects
Updated the path to secure manifest files.	See *2.1 Test layering details* on page 2-21	All revisions
Updated the test execution flow and SPE binary information.	See the following sections: • *2.3 Test execution flow* on page 2-25 • *2.4 Loading test suite binaries* on page 2-28	All revisions

Table A-3 Differences between Issue B and Issue C

Change	Location	Affects
Added information about developer APIs.	See the following sections: • *1.3 Platform Security Architecture APIs* on page 1-13 • *1.4 Test suite* on page 1-15 • *1.6 Directory structure* on page 1-17 • *2.2 Test suite organization* on page 2-23 • *2.3 Test execution flow* on page 2-25	All revisions
Added ITS and PS information.	See the following sections: • *1.2 Abbreviations* on page 1-12 • *1.4 Test suite* on page 1-15	All revisions
Moved information about the test dispatcher to a new section.	See *2.5 Test dispatcher* on page 2-30	All revisions
Updated the test suite summary and debugging details.	See *2.6 Analyzing test run results* on page 2-31	All revisions