

PSA Cryptography API

This document is a general presentation of the PSA Cryptography API. The full specification of the API consists of this document, plus the companion document “Platform Security Architecture — cryptography and keystore interface” which contains a detailed description of the types, functions and macros defined by the API.

Version: beta 1 — 2019-01-21

Introduction

Arm’s 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, based on industry best practice, that allows security to be consistently designed in, at both a hardware and firmware level.

The PSA Cryptographic API (Crypto API) described in this document is an important component of the PSA that provides an interface to modern cryptographic primitives on resource-constrained devices. It constitutes an interface that is easy to comprehend while still providing access to the primitives used in modern cryptography. The interface does not require the user to have access to the key material, instead using opaque key handles.

This document is part of the Platform Security Architecture (PSA) family of specifications. It defines an interface for cryptographic services, including cryptography primitives and a key storage functionality.

This document includes:

- A rationale for the design.
- A description of typical architectures of implementations of this specification.
- A high-level overview of the functionality provided by the interface.
- General considerations for implementers of this specification and for applications that use the interface defined in this specification.

Refer to the companion document “Platform Security Architecture — cryptography and keystore interface” for a detailed definition of the API.

Companion documents will define *profiles* for this specification. A profile is a minimum mandatory subset of the interface that a compliant implementation must provide.

Design goals

Suitable for constrained devices

The interface defined in this document was designed to be suitable for a vast range of devices, from special-purpose cryptographic processors specialized to process data with a built-in key, through constrained devices running custom application code such as microcontrollers, to multi-application devices such as servers. As a consequence, the interface is modular and scalable.

- *Scalable*: you shouldn't pay for functionality that you don't need.
- *Modular*: larger devices implement larger subsets of the same interface, not different interfaces.

Because this specification is designed to be suitable for very constrained devices, including devices where memory is very limited, all operations on unbounded amounts of data allow *multipart* processing if the calculations on the data are performed in a streaming manner. This means that the application does not need to store the whole message in memory at a time.

Memory outside the keystore boundary is meant to be managed by the application. The interface is intended to allow implementations not to retain any data between function calls apart from the content of the keystore and other data that needs to be stored inside the keystore security boundary.

The interface does not expose the representation of keys and intermediate data except when required for interchange. This allows each implementation to choose optimal data representations. Implementations with multiple components are also free to choose which memory area to use for internal data.

A keystore interface

This specification is designed to allow cryptographic operations performed on a key to which the application does not have direct access. Except where required for interchange, applications access all keys indirectly, via a handle. The key material corresponding to that handle may reside inside a security boundary that prevents it from being extracted (except as permitted by a policy defined when the key is created).

Optional isolation

Implementations may optionally isolate the cryptoprocessor from the calling application, and may optionally further isolate multiple calling applications. The interface is designed to allow the implementation to be separated between a frontend and a backend. In an implementation with isolation, the frontend is the part of the implementation that is located in the same isolation boundary as the

application, which the application accesses via function calls, and the backend is the part of the implementation that is located in a different environment which is protected from the frontend. The protection may be provided by a technology such as process isolation in an operating system, partition isolation with a virtual machine or partition manager, physical separation between devices, or any suitable technology. How the frontend and the backend communicate is out of scope of this specification.

In an implementation with isolation, the backend may serve more than one implementation instance. In this case, a single backend communicates with multiple instances of the frontend. The backend must enforce *caller isolation*: it must ensure that assets of one frontend are not visible to any other frontend. How callers are identified is out of scope of this specification. Implementations that provide caller isolation SHALL document how callers are identified. Implementations that provide isolation SHALL document any implementation-specific extension of the API that may allow frontend instances to share data in any form.

In summary, there are three types of implementations:

- No isolation: there is no security boundary between the application and the cryptoprocessor. An example type of implementation with no isolation is a statically or dynamically linked library.
- Cryptoprocessor isolation: there is a security boundary between the application and the cryptoprocessor, but the cryptoprocessor does not communicate with other applications. An example type of implementation with cryptoprocessor isolation is a cryptoprocessor chip that is a companion to an application processor.
- Caller isolation: there are multiple application instances, with a security boundary between the application instances among themselves as well as between the cryptoprocessor and the application instances. An example type of implementation with cryptoprocessor isolation is a cryptography service in a multiprocess environment.

Choice of algorithms

This specification defines a low-level cryptographic interface, where the caller explicitly chooses which algorithm and which security parameters to use. This is necessary to implement protocols that are inescapable in various use cases. The interface is designed to support widespread protocols and data exchange formats, as well as custom ones that applications may need to implement.

As a consequence, all cryptographic functionality operates according to the precise algorithm specified by the caller. (This does not apply to device-internal functionality which does not involve any form of interoperability, such as random number generation.) This specification does not include generic higher-level

interfaces where the implementation chooses the best algorithm for a purpose, but higher-level libraries can be built on top of it.

Another consequence is that this specification permits the use of algorithms, key sizes and other parameters that are known to be insecure, but may be necessary to support legacy protocols or legacy data. Where major weaknesses are known, the algorithm descriptions includes applicable warnings, but the lack of a warning does not and cannot indicate that an algorithm is secure in all circumstances. Application developers should research the security of the algorithms that they plan to use and decide according to their needs.

The interface is designed to facilitate algorithm agility. As a consequence, cryptographic primitives are presented through generic functions, with a parameter indicating the specific choice of algorithm. For example, there is a single function to calculate a message digest, taking a parameter which identifies the specific hash algorithm.

Ease of use

The interface is designed to be as easy to use as possible, given the aforementioned constraints on suitability for varied types of devices and on the freedom to choose algorithms.

In particular, the code flows are designed to reduce the chance of dangerous misuse. The interface is intended to make misuse harder than correct use, and for likely mistakes to result in test failures rather than subtle security issues. Implementations are encouraged to avoid leaking data when a function is called with invalid parameters, to the extent allowed by the C language and by implementation size constraints.

Example use cases

This section lists some of the use cases that were considered when designing this API. This list is not limitative, nor are all implementations required to support all use cases.

Network Security (TLS)

This API should provide everything needed to establish TLS connections on the device side: asymmetric key management inside a key store, symmetric ciphers, MAC, HMAC, message digests, and AEAD.

Secure Storage

This API should provide all primitives related to storage encryption, block- or file-based, with master encryption keys stored inside a key store.

Network Credentials

This API should provide network credential management inside a key store, e.g. for X.509-based authentication or pre-shared keys on enterprise networks.

Device Pairing

This API should provide support for key agreement protocols that are often used for secure pairing of devices over wireless channels, for example pairing an NFC token or a bluetooth device could make use of key agreement protocols upon first use.

Secure Boot

This API should provide primitives for use during firmware integrity and authenticity validation during a secure or trusted boot process.

Attestation

This API should provide primitives used in attestation activities. Attestation is the ability for a device to sign an arbitrary bag of bytes with a device private key and return the result to the caller. Several use cases are attached to this, from attestation of the device state to the ability to generate a key pair and prove that it has been generated inside a secure key store. The API provides access to the algorithms commonly used for attestation.

Factory Provisioning

It is expected that most IoT devices will receive a unique identity during a factory provisioning process or once deployed to the field. This API should provide the APIs necessary for populating a device with keys that represent that identity.

Functionality overview

This section provides a high-level overview of the functionality provided by the interface defined in this specification. Refer to the API definition for a detailed description.

Due to the modularity of the interface, almost every part of the library is optional. The only mandatory function is `psa_crypto_init`.

Library management

Before any use, applications must call `psa_crypto_init` to initialize the library.

Key management

Applications always access keys via a handle. This allows keys to be non-extractable, i.e. an application can perform operations using a key without having access to the key material. Non-extractable keys are bound to the device, can be rate-limited, and can have their usage restricted by policies.

Each key has a *lifetime* that determines when the key material is destroyed. There are two types of lifetimes: volatile and persistent.

Volatile keys

A *volatile* key is destroyed as soon as the application closes the handle to the key. When the application terminates, it conceptually closes all of its key handles. Conceptually, a volatile key is stored in RAM. Volatile keys have the lifetime `PSA_KEY_LIFETIME_VOLATILE`.

To create a volatile key:

1. Call `psa_allocate_key`.
2. Set the key's policy.
3. Provision the key material with `psa_import_key`, `psa_generate_key`, `psa_generator_import_key` or `psa_clone_key`.

To destroy a volatile key, call `psa_close_key` or `psa_destroy_key` (these functions are equivalent when called on a volatile key).

Persistent keys

A *persistent* key exists until it explicitly destroyed with `psa_destroy_key` or until it is wiped by the reset or destruction of the device. Persistent keys may be stored in different storage areas; this is indicated through different lifetime values. This specification defines a lifetime value `PSA_KEY_LIFETIME_PERSISTENT` which corresponds to a default storage area. Implementations may define alternative lifetime values corresponding to different storage areas with different retention policies, or to secure elements with different security characteristics.

To create a persistent key:

1. Call `psa_create_key`, specifying the desired lifetime for the key and the desired persistent identifier. The lifetime value specifies the storage area for the key data and metadata, and the identifier serves as a name. Lifetimes are namespaces for persistent keys: the same key identifier value with distinct lifetime values designates unrelated keys.
2. Set the key's policy.
3. Provision the key material with `psa_import_key`, `psa_generate_key`, `psa_generator_import_key` or `psa_clone_key`.

To release memory resources associated with a key but keep the key in storage, call `psa_close_key`. To access an existing persistent key, call `psa_open_key` with the same lifetime value and the same key identifier as the original call to `psa_create_key`.

To destroy a persistent key, open it (if it isn't already open) and call `psa_destroy_key`.

Recommendations of minimum standards for key management

Most implementations should provide the functions `psa_import_key`. The only exceptions are implementations that only give access to a key or keys that are provisioned by proprietary means and do not allow the main application to use its own cryptographic material.

Most implementations should provide `psa_get_key_information`, `psa_get_key_lifetime` and `psa_get_key_policy` since they are easy to implement and it is difficult to write applications and especially to diagnose issues without being able to check the metadata.

Most implementations should also provide `psa_export_public_key` if they support any asymmetric algorithm, since public-key cryptography often requires delivery of a public key that is associated with a protected private key.

Most implementations should provide `psa_export_key`. However, highly constrained implementations that are designed to work either only with short-term keys (no non-volatile storage) or only with long-term non-extractable keys may omit this function.

Usage policies

All keys have an associated policy that regulates what operations are permitted on the key. This specification defines policies that encode three kinds of attributes:

- The extractable flag `PSA_KEY_USAGE_EXPORT` determines whether the key material can be extracted. The extractable flag is encoded in the usage bitmask which has the type `psa_key_usage_t`.
- The usage flags `PSA_KEY_USAGE_ENCRYPT`, `PSA_KEY_USAGE_SIGN`, etc. determine whether the corresponding operation is permitted on the key. These flags are encoded in the usage bitmask as well.
- In addition to the usage bitmask, a policy specifies which algorithm is permitted with the key. This specification only defines policies that restrict keys to a single algorithm, which is in keeping with common practice and with security good practice.

Most implementations should provide the function `psa_set_key_policy`. Highly constrained implementations that only support slots with preset policies may omit this function.

Symmetric cryptography

This specification defines interfaces for message digests (hash functions), MAC (message authentication codes), symmetric ciphers and authenticated encryption with associated data (AEAD). For each type of primitive, the API includes two standalone functions (compute and verify, or encrypt and decrypt) as well as a series of functions that permit multipart operations.

The standalone functions are:

- `psa_hash_compute` and `psa_hash_compare` to calculate the hash of a message or compare the hash of a message with a reference value.
- `psa_mac_compute` and `psa_mac_verify` to calculate the MAC of a message or compare the MAC with a reference value.
- `psa_cipher_encrypt` and `psa_cipher_decrypt` to encrypt or decrypt a message using an unauthenticated symmetric cipher. The encryption function generates a random IV; to use a deterministic IV (which is not secure in general, but can be secure in some conditions that depend on the algorithm), use the multipart API.
- `psa_aead_encrypt` and `psa_aead_decrypt` to encrypt/decrypt and authenticate a message using an AEAD algorithm. These functions follow the interface recommended by RFC 5116.

Multipart operations

The API provides a multipart interface to hash, MAC, symmetric cipher and AEAD primitives. These interfaces process messages one chunk at a time, with the size of chunks determined by the caller. This allows processing messages that cannot be assembled in memory. The steps to perform a multipart operation are as follows:

1. Allocate an operation object. It is free to use any allocation strategy: stack, heap, static, etc.
2. Initialize the operation object by setting it to zero (either logical zero or all-bits-zero) or by calling one of the applicable macro `PSA_XXX_INIT` or function `psa_xxx_init`.
3. Associate a key with the operation using the applicable function: `psa_hash_setup`, `psa_mac_sign_setup`, `psa_mac_verify_setup`, `psa_cipher_encrypt_setup`, `psa_cipher_decrypt_setup`, `psa_aead_encrypt_setup`, `psa_aead_decrypt_setup`.
4. When encrypting data, generate or set an initialization vector (IV) or nonce or similar initial value such as an initial counter value. When decrypting, set the IV or nonce. For a symmetric cipher, to generate a random IV, which is recommended in most protocols, call `psa_cipher_generate_iv`. To set the IV, call `psa_cipher_set_iv`. For AEAD, call `psa_aead_generate_nonce` or `psa_aead_set_nonce`.

5. Call the applicable update function on successive chunks of the message: `psa_hash_update`, `psa_mac_update` or `psa_cipher_update`.
6. At the end of the message, call the applicable finishing function. There are three kinds of finishing function, depending on what to do with the verification tag.
 - Unauthenticated encryption and decryption does not involve a verification tag. Call `psa_cipher_finish`.
 - To calculate the digest or MAC or authentication tag of a message, call the applicable function to calculate and output the verification tag: `psa_hash_finish`, `psa_mac_sign_finish` or `psa_aead_finish`.
 - To verify the digest or MAC of a message against a reference value or to verify the authentication tag at the end of AEAD decryption, call the applicable function to compare the verification tag with the reference value: `psa_hash_verify`, `psa_mac_verify_finish` or `psa_aead_verify`.

Calling the start/setup function may allocate resources inside the implementation. These resources are freed when calling the associated finishing function. In addition, each family of functions defines a function `psa_xxx_abort` which can be called at any time to free the resources associated with an operation.

Authenticated encryption

Having a multipart interface to authenticated encryption raises specific issues.

Multipart authenticated decryption produces partial results that are not authenticated. Applications must not use or expose partial results of authenticated decryption until `psa_aead_verify` has returned a success status, and must destroy all partial results without revealing them if `psa_aead_verify` returns a failure status. Revealing partial results (directly, or indirectly through the application's behavior) can compromise the confidentiality of all inputs that are encrypted with the same key.

For encryption, some common algorithms cannot be processed in a streaming fashion. For SIV mode, the whole plaintext must be known before the encryption can start; the multipart AEAD API is not meant to be usable with SIV mode. For CCM mode, the length of the plaintext must be known before the encryption can start; the application can call the function `psa_aead_set_lengths` to provide these lengths before providing input.

Key derivation and generators

This specification defines a mechanism for key derivation that allows splitting the output of the derivation into multiple keys as well as non-key outputs.

In an implementation with isolation, the intermediate state of the key derivation is not visible to the caller, and if an output of derivation is a non-exportable key,

then this output cannot be recovered outside the isolation boundary.

Generators

A *generator* is an object that encodes a method to generate a finite stream of bytes. This data stream is computed by the cryptoprocessor and extracted in chunks. The intent of generators is that if two generators are constructed with the same parameters then they will produce the same outputs.

Some examples of generators are:

- A pseudorandom generator, initialized with a seed and other parameters.
- A key derivation function, initialized with a secret, a salt and other parameters.
- A key agreement function, initialized with a public key (peer key), a key pair (own key) and other parameters.

The lifecycle of a generator is as follows:

1. Setup: construct the object and set its parameters. The setup phase determines the generator's capacity, which is the length of the generated stream, i.e. the maximum number of bytes that can be generated with this generator.
2. Generate: read bytes from the stream defined by the generator. This can be done any number of times until the stream is exhausted because its capacity has been reached. Each generation step can either be used to populate a key object (`psa_generator_import_key`), or to read some bytes and extract them as cleartext (`psa_generator_read`).
3. Terminate: clear the generator object and release associated resources (`psa_generator_abort`).

A generator cannot be rewinded. Once a part of the stream has been read, it cannot be read again. This ensures that the same part of the generator output will not be used from different purposes.

Key derivation function

This specification defines functions to set up a key derivation. A key derivation consists of two parts:

1. Input collection. This is sometimes known as *extraction*: the operation “extracts” information from the inputs to generate a pseudorandom intermediate secret value.
2. Output generation. This is sometimes known as *expansion*: the operation “expands” the intermediate secret value to the desired output length.

The API uses a generator object to store the state of a key derivation operation. To perform a key derivation:

1. Initialize a generator object to zero or to `PSA_CRYPT0_GENERATOR_INIT`.
2. Call `psa_key_derivation_setup` to select a key derivation algorithm.
3. Call the functions `psa_key_derivation_input_bytes`, `psa_key_derivation_input_key` and `psa_key_agreement` to provide the inputs to the key derivation algorithm. Many key derivation algorithm take multiple inputs; the “step” parameter to these functions indicates which input is being passed.
4. Call `psa_generator_import_key` to create a derived key, or `psa_generator_read` to export the derived data. These functions may be called multiple times to read successive output from the key derivation.
5. Call `psa_generator_abort` to release the generator memory.

Here is an example of a use case where a master key is used to generate both a message encryption key and an IV for the encryption, and the derived key and IV are then used to encrypt a message.

1. Allocate a key slot for the derived message encryption key and set its policy.
2. Derive the message encryption material from the master key.
 1. Initialize a generator object to zero or to `PSA_CRYPT0_GENERATOR_INIT`.
 2. Call `psa_key_derivation_setup` with `PSA_ALG_HKDF` as the algorithm.
 3. Call `psa_key_derivation_input_key` with the step `PSA_KDF_STEP_SECRET` and the master key.
 4. Call `psa_key_derivation_input_bytes` with the step `PSA_KDF_STEP_INFO` and a public value that uniquely identifies the message.
 5. Call `psa_generator_import_key` to create the derived message key.
 6. Call `psa_generator_read` to generate the derived IV.
 7. Call `psa_generator_abort` to release the generator memory.
3. Encrypt the message with the derived material.
 1. Call `psa_cipher_encrypt_setup` with the derived encryption key.
 2. Call `psa_cipher_set_iv` using the derived IV retrieved above.
 3. Call `psa_cipher_update` one or more times to encrypt the message.
 4. Call `psa_cipher_finish` at the end of the message.
4. Call `psa_destroy_key` to clear the generated key.

Asymmetric cryptography

The asymmetric cryptography part of this interface defines functions for asymmetric encryption, asymmetric signature and two-way key agreement.

Asymmetric encryption

Asymmetric encryption is provided through the functions `psa_asymmetric_encrypt` and `psa_asymmetric_decrypt`.

Hash-and-sign

The signature and verification functions `psa_asymmetric_sign` and `psa_asymmetric_verify` take a hash as one of their inputs. This hash should be calculated with `psa_hash_setup`, `psa_hash_update` and `psa_hash_finish` before calling `psa_asymmetric_sign` or `psa_asymmetric_verify`. To determine which hash algorithm to use, call the macro `PSA_ALG_SIGN_GET_HASH` on the corresponding signature algorithm.

Key agreement

This specification defines two functions for a Diffie-Hellman-style key agreement where each party combines its own private key with the peer's public key. The recommended function is `psa_key_agreement`, which calculates a shared secret and passes it as one of the inputs to a key derivation function. In case an application needs direct access to the shared secret, it can call `psa_key_agreement_raw_shared_secret`; note that in general the shared secret is not directly suitable for use as a key because it is biased.

Randomness and key generation

It is strongly recommended that implementations include a random generator consisting of a cryptographically secure pseudo-random generator (CSPRNG) which is adequately seeded with a cryptographic-quality hardware entropy source, commonly referred to as a true random number generator (TRNG). Constrained implementations may omit the random generation functionality if they do not implement any algorithm that requires randomness internally and they do not provide a key generation functionality — for example a special-purpose component for signature verification.

Applications should use `psa_generate_key`, `psa_encrypt_generate_iv` or `psa_aead_generate_iv` to generate suitably-formatted random data as applicable. In addition, the API includes a function `psa_generate_random` to generate and extract arbitrary random data.

Future additions

We plan to cover the following features in future drafts or future editions of this specification:

- Single-shot functions for symmetric operations.
- Multi-part operations for hybrid cryptography: hash-and-sign (e.g. for EdDSA), hybrid encryption (e.g. for ECIES).
- Key exchange and a more general interface to key derivation. This would enable deriving a non-extractable session key from non-extractable secrets without leaking the intermediate material.

- Key wrapping mechanisms, to extract and import keys in a protected form (encrypted and authenticated).
- Key discovery and slot discovery mechanisms. This would enable locating a key through its name or attributes rather than having to hard-code slot numbers, and finding a slot to contain a key prior to creating the key.
- An ownership and access control mechanism allowing a multi-client implementation to have privileged clients that are able to manage keys of other clients.

Sample architectures

This section describes some possible architectures of implementations of the interface described in this specification. This list of architectures is not limitative and this section is entirely non-normative.

Single-partition architecture

In this architecture, there is no security boundary inside the system. The application code may access all the system memory, including the memory used by the cryptographic services described by this specification. Thus this architecture provides no isolation.

This architecture does not conform to the Arm Platform Security Architecture specification. However, it may be useful to provide cryptographic services using the same interface even on devices that cannot support any security boundary. Therefore, while this architecture is not the primary design goal of the API defined in the present specification, it is supported.

In this case, the functions in this specification simply execute the underlying algorithmic code. Security checks can be kept to a minimum since the cryptoprocessor cannot defend against a malicious application. Key import and export copy data inside the same memory space.

This architecture also describes a subset of some larger systems where the cryptographic services are implemented inside a high-security partition, separate from the code of the main application, but it shares this high-security partition with other platform security services.

Cryptographic token and single-application processor

This example system is composed of two partitions: one partition is a cryptoprocessor, and the other partition runs an application. There is a security boundary between the two partitions, so that the application cannot access the

cryptoprocessor except through its public interface. Thus this architecture provides cryptoprocessor isolation. The cryptoprocessor includes some nonvolatile storage, a TRNG, and possibly some cryptographic accelerators.

There are multiple possible physical realizations: the cryptoprocessor may be a separate chip, a separate processor on the same chip, or a logical partition using a combination of hardware and software to provide the isolation. These realizations are functionally equivalent in terms of the offered software interface, but they would typically offer different levels of security guarantees.

The PSA crypto API in the application processor consists of a thin layer of code that translates function calls to remote procedure calls in the cryptoprocessor. All cryptographic computations are therefore performed inside the cryptoprocessor. Non-volatile keys are stored inside the cryptoprocessor.

Cryptoprocessor with no key storage

Like the previous one, this example system is composed of two partitions separated by a security boundary. Thus this architecture also provides cryptoprocessor isolation. Unlike the previous architecture, in this case, the cryptoprocessor does not have any secure persistent storage that could be used to store application keys.

If the cryptoprocessor is not capable of storing any cryptographic material, then there is little use for a separate cryptoprocessor, since all data would have to be imported by the application.

The cryptoprocessor can provide useful services if it is able to store at least one key. This may be a hardware unique key that is burnt to one-time programmable memory during the manufacturing of the device. This key may be used for one or more purposes including:

- Encrypt and authenticate data whose storage is delegated to the application processor.
- Communicate with a paired device.
- Allow the application to perform operations with keys that are derived from the hardware unique key.

Multi-client cryptoprocessor

This is an expanded variant of the cryptographic token plus application architecture. In this variant, the cryptoprocessor serves multiple applications that are mutually untrustworthy. This architecture provides caller isolation.

In this architecture, API calls are translated to remote procedure calls which encode the identity of the client application. The cryptoprocessor carefully

segments its internal storage to ensure that a client's data is never leaked to another client.

Multi-cryptoprocessor architecture

In this example, the system includes multiple cryptoprocessors. Some reasons to have multiple cryptoprocessors include:

- Different compromises between security and performance for different keys. Typically this means a cryptoprocessor running on the same hardware as the main application and processing short-term secrets, and a secure element or similar separate chip that retains long-term secrets.
- Independent provisioning of certain secrets.
- A combination of a non-removable cryptoprocessor and removable ones (e.g. a smartcard or HSM).

The keystore implementation needs to dispatch each request to the correct processor. All requests involving a non-extractable key must be processed in the cryptoprocessor that holds that key. Other requests may target a cryptoprocessor based on parameters supplied by the application or based on considerations such as performance inside the implementation. A typical choice for dispatch is for the implementation to define ranges of key slot numbers, such that each range corresponds to one of the cryptoprocessors.

Library conventions

Error handling

Almost all functions return a status indication of type `psa_status_t`. This is an enumeration of integer values, with 0 (`PSA_SUCCESS`) conveying successful operation and other values indicating errors. The exception is data structure accessor functions that cannot fail: such functions may return `void` or a data value.

All function calls must be implemented atomically:

- When a function returns a type other than `psa_status_t`, the requested action has been carried out.
- When a function returns the status `PSA_SUCCESS`, the requested action has been carried out.
- When a function returns another status of type `psa_status_t`, no action has been carried out. The content of output parameters is undefined, but otherwise the state of the system has not changed except as described below.

Generally speaking, functions that modify the system state (modifying the content of a key slot or its metadata) must leave the system state unchanged if they return an error code. However, there are a few exceptions to this general principle in exceptional conditions:

- The status `PSA_ERROR_BAD_STATE` indicates that a supplied parameter was not in a valid state for the requested action. The corresponding object may have been modified by the call and must not be used for any further action except to abort the corresponding object.
- The status `PSA_ERROR_INSUFFICIENT_CAPACITY` indicates that a generator has reached its maximum capacity. The generator object may have been modified by the call and any further attempt to read from the generator will return `PSA_ERROR_INSUFFICIENT_CAPACITY`.
- The status `PSA_ERROR_COMMUNICATION_FAILURE` indicates that the communication between the application and the cryptoprocessor has broken down. In this case, it may be impossible to know whether the action has been carried out. Upon detection of a communication failure, the cryptoprocessor must either finish carrying out the request or roll back to the original state, but the application may not be able to find out which of these two possibilities happened.
- The statuses `PSA_ERROR_STORAGE_FAILURE`, `PSA_ERROR_HARDWARE_FAILURE` and `PSA_ERROR_TAMPERING_DETECTED` may indicate data corruption in the system state. Thus, when a function returns one of these statuses, the system state may have changed compared to before the function call, even though the function call failed.
- Some system state cannot be rolled back, for example the internal state of the random number generator, or the content of logs if the implementation keeps access logs.

Unless otherwise documented, the content of output parameters is not defined when a function returns a status other than `PSA_SUCCESS`. Implementations should set output parameters to safe defaults to avoid leaking confidential data and to limit the risks in case an application does not properly handle all errors.

Parameter conventions

Pointer conventions

Unless explicitly stated in the documentation of a function, all pointers must be valid pointers to an object of the specified type.

A parameter is considered a *buffer* if it points to an array of bytes. A buffer parameter always has the type `uint8_t *` or `const uint8_t *`, and always has an associated parameter indicating the size of the array. Note that a parameter of type `void *` is never considered a buffer.

All parameters of pointer type must be valid, non-null pointers unless the pointer

is to a buffer of length 0 or the function’s documentation explicitly describes the behavior when the pointer is null. Implementations where a null pointer dereference usually aborts the application, passing `NULL` as a function parameter where a null pointer is not allowed should abort the caller in the habitual manner.

Pointers to input parameters may be in read-only memory. Output parameters must be in writable memory. Output parameters that are not buffers must also be readable, and the implementation must be able to write to a non-buffer output parameter and read back the same value, as explained in the section “Stability of parameters”.

Input buffer sizes

For input buffers, the parameter convention is:

- `const uint8_t *foo`: pointer to the first byte of the data. The pointer may be invalid if the buffer size is 0.
- `size_t foo_length`: size of the buffer in bytes.

The interface never uses input-output buffers.

Output buffer sizes

For output buffers, the parameter convention is:

- `uint8_t *foo`: pointer to the first byte of the data. The pointer may be invalid if the buffer size is 0.
- `size_t foo_size`: the size of the buffer in bytes.
- `size_t *foo_length`: on successful return, contains the length of the output in bytes.

The content of the data buffer and of `*foo_length` on error is unspecified unless explicitly mentioned in the function description. They may be unmodified or set to a safe default. On successful completion, the content of the buffer between the offsets `*foo_length` and `foo_size` is also unspecified.

Functions return `PSA_ERROR_BUFFER_TOO_SMALL` if the buffer size is insufficient to carry out the requested operation. The interface defines macros to calculate a sufficient buffer size for each operation that has an output buffer. These macros return compile-time constants if their arguments are compile-time constants, so they are suitable for static or stack allocation. Refer to individual functions’ documentation for the associated output size macro.

Some functions always return exactly as much data as the size of the output buffer. In this case, the parameter convention changes to:

- `uint8_t *foo`: pointer to the first byte of the output. The pointer may be invalid if the buffer size is 0.
- `size_t foo_length`: the number of bytes to return in `foo` if successful.

Overlap between parameters

Output parameters that are not buffers may not overlap with any input buffer or with any other output parameter. Otherwise the behavior is undefined.

Output buffers may overlap with input buffers. If this happens, the implementation must return the same result as if the buffers did not overlap. In other words, the implementation must behave as if it had copied all the inputs into temporary memory, as far as the result is concerned. However application developers should note that overlap between parameters may affect the performance of a function call. Overlap may also affect the security of how memory is managed if the buffer is located in memory that the caller shares with another security context, as described in the section “Stability of parameters”.

Stability of parameters

In some environments, it is possible for the content of a parameter to change while a function is executing. It may also be possible for the content of an output parameter to be read before the function terminates. This can happen if the application is multithreaded. In some implementations, memory can be shared between security contexts, for example, between tasks in a multitasking operating system, or between a user land task and the kernel, or between the non-secure world and the secure world of a trusted execution environment. This section describes what implementations need or need not guarantee in such cases.

Parameters that are not buffers are assumed to be under the caller’s full control. In a shared memory environment, this means that the parameter must be in memory that is exclusively accessible by the application. In a multithreaded environment, this means that the parameter may not be modified during the execution and the value of an output parameter is undetermined until the function returns. The implementation may read an input parameter that is not a buffer multiple times and expect to read the same data. The implementation may write to an output parameter that is not a buffer and expect to read back the value that it last wrote. The implementation has the same permissions on buffers that overlap with a buffer in the opposite direction.

In an environment with multiple threads or with shared memory, the implementation shall access non-overlapping buffer parameters carefully in order to prevent any unsafety if the content of the buffer is modified or observed during the execution of the function. In an input buffer that does not overlap with an output buffer, the implementation shall read each byte of the input at most once. The implementation shall not read from an output buffer that does not overlap with an input buffer. Additionally, the implementation shall not write data to a non-overlapping output buffer if this data is potentially confidential and the implementation has not yet verified that outputting this data is authorized.

Key types and algorithms

Types and cryptographic keys and cryptographic algorithms are encoded separately. Each is encoded using an integral type, respectively `psa_key_type_t` and `psa_algorithm_t`.

There is some overlap in the information conveyed through keys and algorithms. Both types include enough information so that the meaning of an algorithm type value does not depend on what type of key it is used with and vice versa. However, the particular instance of an algorithm may depend on the key type. For example, the algorithm `PSA_ALG_GCM` can be instantiated as any AEAD algorithm using the GCM mode over a block cipher; the underlying block cipher is determined by the key type.

Key types do not encode the key size. For example AES-128, AES-192 and AES-256 share a key type `PSA_KEY_TYPE_AES`.

Structure of key and algorithm types

Both types use a partial bitmask structure which allows analyzing and building values from parts. However the interface defines constants so that applications do not need to depend on the encoding and an implementation may care about the encoding only for code size optimization.

The encodings follows a few conventions:

- The highest bit is a vendor flag. Values with this bit clear are reserved for values defined by this specification, and values with this bit set will not be defined by this specification.
- The next few highest bits indicate the corresponding algorithm category: hash, MAC, symmetric cipher, asymmetric encryption, etc.
- The following bits identify a family of algorithms in a category-dependent manner.
- In some categories and algorithm families, the lowest-order bits indicate a variant in a systematic way. For example, algorithm families that are parametrized around a hash function encode the hash in the 8 lowest bits.

Concurrent calls

In some environments, it is possible for an application to make calls to the PSA crypto API in separate threads. In such an environment, concurrent calls SHALL be performed correctly, as if the calls had been executed in sequence, provided that they obey the following constraints:

- There must not be any overlap between an output parameter of one call and an input or output parameter of another call. (Overlap between input parameters is permitted.)

- If a call modifies a key slot, then no other call must modify or use that key slot. *Using*, in this context, includes all functions of multipart operations using the key. (Concurrent calls that merely use the same key are permitted.)
- Concurrent calls may not use the same operation or generator object.

If any of these constraints is violated, the behavior is undefined.

Individual implementations may provide additional guarantees.

Implementation considerations

Implementation-specific aspects of the interface

Implementation profile

Implementations may implement a subset of the API and a subset of the available algorithms. The implemented subset is known as the implementation's profile. The documentation of each implementation SHALL document what profile it implements. Companion documents to this specification will define standard profiles.

Implementation-specific types

This specification defines some platform-specific types to represent data structures whose content depends on the implementation. These types are C `struct` types. In the associated header files, `crypto.h` declares the `struct` tags and `crypto_struct.h` provides a definition for the structures.

Implementation-specific macros

Some macros compute a result based on an algorithm or a key type. This specification provides a sample implementation of these macros which works for all standard types. If an implementation defines vendor-specific algorithms or key types, it must provide an implementation of such macros that takes all relevant algorithms and types into account. Conversely, an implementation that does not support a certain algorithm or key type may define such macros in a simpler way that does not take unsupported argument values into account.

Some macros define the minimum sufficient output buffer size for certain functions. In some cases, implementations are allowed to require a buffer size that is larger than the theoretical minimum. Implementations SHALL define minimum-size macros in such a way as to guarantee that a buffer of the resulting size is sufficient for the output of the corresponding function. Refer to each macro's documentation for the applicable requirements.

Porting to a platform

Platform assumptions

This specification is designed for a C89 platform. The interface is defined in terms of C macros, functions and objects.

This specification assumes 8-bit bytes. In this specification, “byte” and “octet” are used synonymously.

Platform-specific types

The specification makes use of some platform-specific types which should be defined in `crypto_platform.h` (possibly via a header included by this file). `crypto_platform.h` must define the following types:

- `uint8_t`, `uint16_t`, `uint32_t`: unsigned integer types with 8, 16 and 32 value bits respectively. These may be the types defined by the C99 header `stdint.h`.
- `psa_key_handle_t`: an unsigned integer type of the implementation’s choice.

Cryptographic hardware support

Implementations are encouraged to make use of hardware accelerators where available. A future version of this specification will define a function interface to call drivers for hardware accelerators and external cryptographic hardware.

Security requirements and recommendations

Error detection

Implementations that provide isolation between the caller and the cryptography processing environment SHALL validate parameters to ensure that the cryptography processing environment is protected from attacks caused by passing invalid parameters.

Even implementations that do not provide isolation should strive to detect bad parameters and fail safe as much as possible.

Memory cleanup

Implementations SHALL wipe all sensitive data from memory when it is no longer used. Implementations should wipe sensitive data as soon as possible. In any case, all temporary data (such as stack buffers) used during the execution of a function shall be wiped before the function returns, and all data associated with an object (such as a multipart operation) shall be wiped at the latest

when the object becomes inactive (for example, when a multipart operation is aborted).

The rationale for this non-functional requirement is to minimize the impact if the system is compromised. If sensitive data is wiped immediately after use, a data leak only leaks data that is currently in use, but does not compromise past data.

Safe outputs on error

Implementations SHALL ensure that no confidential data is written to output parameters before validating that the disclosure of this confidential data is authorized. This requirement is especially important on implementations where the caller may share memory with another security context as described in the section “Stability of parameters”.

In most cases, this specification does not define the content of output parameters when an error occurs. Implementations should ensure that the content of output parameters is as safe as possible in case it ends up being used due to an application flaw or a data leak. In particular, implementations should avoid placing partial output in output buffers if an action is interrupted. The definition of “safe” is left up to each implementation as different environments may require different compromises between implementation complexity, overall robustness and performance. Some common strategies include leaving output parameters unchanged in case of errors, or zeroing them out.

Attack resistance

Cryptographic code tends to manipulate high-value secrets from which other secrets can be unlocked. As such it is a high-value target for attacks. A vast body of literature exists on attack types such as side channel attacks and glitch attacks. Typical side channels include timing, cache access patterns, branch prediction access patterns, power consumption, radio emissions and more.

This specification does not place any particular requirement regarding attack resistance. Implementers should consider the attack resistance that is expected in each use case and design their implementation accordingly. Security standards that define targets for attack resistance may be applicable in certain use cases.

Other implementation considerations

Philosophy of resource management

This specification allows most functions to return `PSA_ERROR_INSUFFICIENT_MEMORY`. This gives implementations the freedom to manage memory as they please.

Nonetheless the interface is designed to allow conservative strategies for memory management. In particular, an implementation may avoid dynamic memory allocation altogether by obeying certain restrictions:

- Pre-allocate memory for a predefined number of key slots, each with sufficient memory for all key types that can be stored in that slot.
- For multipart operations, in an implementation without isolation, place all the data that needs to be carried over from one step to the next in the operation object. The application is then fully in control of how memory is allocated for the operation.
- In an implementation with isolation, pre-allocate memory for a predefined number of operations inside the cryptoprocessor.

Usage considerations

Security recommendations

Always check for errors

Most functions in this API can return errors. All functions that can fail have the return type `psa_status_t`. A few functions cannot fail, and thus return `void` or some other type.

If an error occurs, unless otherwise specified, the content of output parameters is undefined and must not be used.

Some common causes of errors include:

- In implementations where the keys are stored and processed in a separate environment from the application, all functions that need to access the cryptography processing environment may fail due to an error in the communication between the two environments.
- If an algorithm is implemented with a hardware accelerator which is logically separate from the application processor, this accelerator may fail even when the application processor keeps running normally.
- All functions may fail due to a lack of resources, although some implementations may guarantee that certain functions always have sufficient memory.
- All functions that access persistent keys may fail due to a storage failure.
- All functions that require randomness may fail due to a lack of entropy. Implementations are encouraged to seed the random generator with sufficient entropy during the execution of `psa_crypto_init`; however some security standards require periodic reseeding from a hardware random generator which can fail.

Shared memory and concurrency

Some environment allow applications to be multithreaded. In some environments, applications may share memory with a different security context. In such environments, applications must be written carefully to avoid data corruption or leakage. This specification requires the application to obey certain constraints.

In general, this API allows either one writer or any number of simultaneous readers on any given object. In other words, if two or more calls access the same object concurrently, the behavior is well-defined only if all the calls are only reading from the object and do not modify it. Read accesses include reading memory via input parameters and reading key store content by using a key. For more details, refer to the section “Concurrent calls”.

If an application shared memory with another security contexts, it may pass shared memory blocks as input buffers or output buffers, but not as non-buffer parameters. For more details from the implementation’s perspective, refer to the section “Stability of parameters”.

Cleaning up after use

In order to minimize the impact if the system is compromised, applications should wipe all sensitive data from memory when it is no longer used. This way, a data leak only leaks data that is currently in use, but does not compromise past data.

Wiping sensitive data includes:

- Clearing temporary buffers in the stack or on the heap.
- Aborting operations if they will not be finished.
- Destroying key slots that are no longer used.