PSA Cryptography API 1.0

Document number:	IHI 0086
Release Quality:	Final
Issue Number:	1
Confidentiality:	Non-confidential
Date of Issue:	27/08/2020

Contents

About this document

Release information

The change history table lists the changes that have been made to this document.

Date	Version	Confidentiality	Change
January 2019	1.0 Beta 1	Non-confidential	First public beta release.
February 2019	1.0 Beta 2	Non-confidential	Update for release with other PSA Dev API specifications.
May 2019	1.0 Beta 3	Non-confidential	Update for release with other PSA API specifications.
February 2020	1.0 Final	Non-confidential	1.0 API finalized.
August 2020	1.0.1 Final	Non-confidential	Update to fix errors and provide clarifications.

The detailed changes in each release are described in *Document change history* on page 242.

PSA Cryptography API

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

Arm document reference: LES-PRE-21585 version 4.0

References

This document refers to the following documents.

Table 1 Arm documents referenced by this document

Ref	Document Number	Title
[PSA-ITS]	ARM IHI 0087	*PSA Storage API*. https://developer.arm.com/architectures/security-architectures/platform-security-architecture/documentation

Table 2 Other documents referenced by this document

Ref	Title
[CHACHA20]	Bernstein, D., *ChaCha, a variant of Salsa20*, January 2008. http://cr.yp.to/chacha/chacha-20080128.pdf
[Curve25519]	Bernstein et al., *Curve25519: new Diffie-Hellman speed records*, LNCS 3958, 2006. https://www.iacr.org/archive/pkc2006/39580209/39580209.pdf
[Curve448]	Hamburg, *Ed448-Goldilocks, a new elliptic curve*, NIST ECC Workshop, 2015. https://eprint.iacr.org/2015/625.pdf
[FIPS180-4]	NIST, *FIPS Publication 180-4: Secure Hash Standard (SHS)*, August 2015. https://doi.org/10.6028/NIST.FIPS.180-4
[FIPS186-4]	NIST, *FIPS Publication 186-4: Digital Signature Standard (DSS)*, July 2013. https://doi.org/10.6028/NIST.FIPS.186-4
[FIPS197]	NIST, *FIPS Publication 197: Advanced Encryption Standard (AES)*, November 2001. https://doi.org/10.6028/NIST.FIPS.197
[FIPS202]	NIST, *FIPS Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, August 2015. https://doi.org/10.6028/NIST.FIPS.202
[FRP]	Agence nationale de la sécurité des systèmes d'information, *Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française*, 21 November 2011. https://www.ssi.gouv.fr/agence/rayonnement-scientifique/publications-scientifiques/articles-ouvrages-actes
[IEEE-XTS]	IEEE, *1619-2018 - IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*, January 2019. https://ieeexplore.ieee.org/servlet/opac?punumber=8637986
[IETF-SM3]	IETF, *The SM3 Cryptographic Hash Function*, November 2017. https://tools.ietf.org/id/draft-oscca-cfrg-sm3-02.html
[IETF-SM4]	IETF, *The SM4 Blockcipher Algorithm And Its Modes Of Operations*, April 2018. https://tools.ietf.org/html/draft-ribose-cfrg-sm4-10
[ISO10118]	ISO/IEC, *ISO/IEC 10118-3:2018 IT Security techniques — Hash-functions — Part 3: Dedicated hash-functions*, October 2018. https://www.iso.org/standard/67116.html

Table 2 (continued)

Ref	Title
[ISO9797]	ISO/IEC, *ISO/IEC 9797-1:2011 Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher*, March 2011. https://www.iso.org/standard/50375.html
[NTT-CAM]	NTT Corporation and Mitsubishi Electric Corporation, *Specification of Camellia — a 128-bit Block Cipher*, September 2001. https://info.isl.ntt.co.jp/crypt/eng/camellia/specifications
[PRC-SM3]	Standardization Administration of the People's Republic of China, *GB/T 32905-2016: Information security techniques — SM3 cryptographic hash algorithm*, August 2016. http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=45B1A67F20F3BF339211C391E9278F5E
[PRC-SM4]	Standardization Administration of the People's Republic of China, *GB/T 32907-2016: Information security technology — SM4 block cipher algorithm*, August 2016. http://www.gb688.cn/bzgk/gb/newGbInfo?hcno=7803DE42D3BC5E80B0C3E5D8E873D56A
[RFC1319]	IETF, *The MD2 Message-Digest Algorithm*, April 1992. https://tools.ietf.org/html/rfc1319.html
[RFC1320]	IETF, *The MD4 Message-Digest Algorithm*, April 1992. https://tools.ietf.org/html/rfc1320.html
[RFC1321]	IETF, *The MD5 Message-Digest Algorithm*, April 1992. https://tools.ietf.org/html/rfc1321.html
[RFC2104]	IETF, *HMAC: Keyed-Hashing for Message Authentication*, February 1997. https://tools.ietf.org/html/rfc2104.html
[RFC2315]	IETF, *PKCS #7: Cryptographic Message Syntax Version 1.5*, March 1998. https://tools.ietf.org/html/rfc2315.html
[RFC3279]	IETF, *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, April 2002. https://tools.ietf.org/html/rfc3279.html
[RFC3610]	IETF, *Counter with CBC-MAC (CCM)*, September 2003. https://tools.ietf.org/html/rfc3610
[RFC3713]	IETF, *A Description of the Camellia Encryption Algorithm*, April 2004. https://tools.ietf.org/html/rfc3713
[RFC4279]	IETF, *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*, December 2005. https://tools.ietf.org/html/rfc4279.html
[RFC5116]	IETF, *An Interface and Algorithms for Authenticated Encryption*, January 2008. https://tools.ietf.org/html/rfc5116.html
[RFC5246]	IETF, *The Transport Layer Security (TLS) Protocol Version 1.2*, August 2008. https://tools.ietf.org/html/rfc5246.html
[RFC5639]	IETF, *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, March 2010. https://tools.ietf.org/html/rfc5639.html

Table 2 (continued)

Ref	Title
[RFC5869]	IETF, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*, May 2010. https://tools.ietf.org/html/rfc5869.html
[RFC5915]	IETF, *Elliptic Curve Private Key Structure*, June 2010. https://tools.ietf.org/html/rfc5915.html
[RFC6979]	IETF, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, August 2013. https://tools.ietf.org/html/rfc6979.html
[RFC7539]	IETF, *ChaCha20 and Poly1305 for IETF Protocols*, May 2015. https://tools.ietf.org/html/rfc7539.html
[RFC7748]	IETF, *Elliptic Curves for Security*, January 2016. https://tools.ietf.org/html/rfc7748.html
[RFC7919]	IETF, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, August 2016. https://tools.ietf.org/html/rfc7919.html
[RFC8017]	IETF, *PKCS #1: RSA Cryptography Specifications Version 2.2*, November 2016. https://tools.ietf.org/html/rfc8017.html
[RIPEMD]	Dobbertin, Bosselaers and Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD*, April 1996. https://homes.esat.kuleuven.be/~bosselae/ripemd160.html
[SEC1]	Standards for Efficient Cryptography, *SEC 1: Elliptic Curve Cryptography*, May 2009. https://www.secg.org/sec1-v2.pdf
[SEC2]	Standards for Efficient Cryptography, *SEC 2: Recommended Elliptic Curve Domain Parameters*, January 2010. https://www.secg.org/sec2-v2.pdf
[SEC2v1]	Standards for Efficient Cryptography, *SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0*, September 2000. https://www.secg.org/SEC2-Ver-1.0.pdf
[SP800-38A]	NIST, *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, December 2001. https://doi.org/10.6028/NIST.SP.800-38A
[SP800-38B]	NIST, *NIST Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication*, May 2005. https://doi.org/10.6028/NIST.SP.800-38B
[SP800-38D]	NIST, *NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*, November 2007. https://doi.org/10.6028/NIST.SP.800-38D
[SP800-56A]	NIST, *NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*, April 2018. https://doi.org/10.6028/NIST.SP.800-56Ar3
[SP800-67]	NIST, *NIST Special Publication 800-67: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher*, November 2017. https://doi.org/10.6028/NIST.SP.800-67r2

Table 2 (continued)

Ref	Title
[X9-62]	ANSI, *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. https://standards.globalspec.com/std/1955141/ANSI%20X9.62

Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
AEAD	See *Authenticated Encryption with Associated Data*.
Algorithm	A finite sequence of steps to perform a particular operation. In this specification, an algorithm is a *cipher* or a related function. Other texts call this a cryptographic mechanism.
API	Application Programming Interface.
Asymmetric	See *Public-key cryptography*.
Authenticated Encryption with Associated Data (AEAD)	A type of encryption that provides confidentiality and authenticity of data using *symmetric* keys.
Byte	In this specification, a unit of storage comprising eight bits, also called an octet.
Cipher	An algorithm used for encryption or decryption with a *symmetric* key.
Cryptoprocessor	The component that performs cryptographic operations. A cryptoprocessor might contain a *keystore* and countermeasures against a range of physical and timing attacks.
Hash	A cryptographic hash function, or the value returned by such a function.
HMAC	A type of *MAC* that uses a cryptographic key with a *hash* function.
IMPLEMENTATION DEFINED	Behavior that is not defined by the architecture, but is defined and documented by individual implementations.
Initialization vector (IV)	An additional input that is not part of the message. It is used to prevent an attacker from making any correlation between cipher text and plain text. This specification uses the term for such initial inputs in all contexts. For example, the initial counter in CTR mode is called the IV.
IV	See *Initialization vector*.
KDF	See *Key Derivation Function*.
Key agreement	An algorithm for two or more parties to establish a common secret key.

Table 3 (continued)

Term	Meaning
Key Derivation Function (KDF)	Key Derivation Function. An algorithm for deriving keys from secret material.
Key identifier	A reference to a cryptographic key. Key identifiers in the PSA Crypto API are 32-bit integers.
Key policy	Key metadata that describes and restricts what a key can be used for.
Key size	The size of a key as defined by common conventions for each key type. For keys that are built from several numbers of strings, this is the size of a particular one of these numbers or strings. This specification expresses key sizes in bits.
Key type	Key metadata that describes the structure and content of a key.
Keystore	A hardware or software component that protects, stores, and manages cryptographic keys.
Lifetime	Key metadata that describes when a key is destroyed.
MAC	See *Message Authentication Code*.
Message Authentication Code (MAC)	A short piece of information used to authenticate a message. It is created and verified using a *symmetric* key.
Message digest	A *hash* of a message. Used to determine if a message has been tampered.
Multi-part operation	An *API* which splits a single cryptographic operation into a sequence of separate steps.
Non-extractable key	A key with a *key policy* that prevents it from being read by ordinary means.
Nonce	Used as an input for certain *AEAD* algorithms. Nonces must not be reused with the same key because this can break a cryptographic protocol.
Persistent key	A key that is stored in protected non-volatile memory.
PSA	Platform Security Architecture
Public-key cryptography	A type of cryptographic system that uses key pairs. A keypair consists of a (secret) private key and a public key (not secret). A public key cryptographic algorithm can be used for key distribution and for digital signatures.
Salt	Used as an input for certain algorithms, such as key derivations.
Signature	The output of a digital signature scheme that uses an *asymmetric* keypair. Used to establish who produced a message.
Single-part function	An *API* that implements the cryptographic operation in a single function call.
SPECIFICATION DEFINED	Behavior that is defined by this specification.
Symmetric	A type of cryptographic algorithm that uses a single key. A symmetric key can be used with a block cipher or a stream cipher.

Table 3 (continued)

Term	Meaning
Volatile key	A key that has a short lifespan and is guaranteed not to exist after a restart of an application instance.

Potential for change

The contents of this specification are stable for version 1.0.

The following may change in updates to the version 1.0 specification:

- Small optional feature additions.
- Clarifications.

Significant additions, or any changes that affect the compatibility of the interfaces defined in this specification will only be included in a new major or minor version of the specification.

Conventions

Typographical conventions

The typographical conventions are:

italic Introduces special terminology, and denotes citations.

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

SMALL CAPITALS

Used for some common terms such as IMPLEMENTATION DEFINED.

Used for a few terms that have specific technical meanings, and are included in the *Terms and abbreviations*.

Red text Indicates an open issue.

Blue text Indicates a link. This can be

- A cross-reference to another location within the document
- A URL, for example http://infocenter.arm.com

Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by `0x`.

In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`. To improve readability, long numbers can be written with an underscore separator between every four characters, for example `0xFFFF_0000_0000_0000`. Ignore any underscores when interpreting the value of a number.

Pseudocode descriptions

This book uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font. The pseudocode language is described in the Arm Architecture Reference Manual.

Assembler syntax descriptions

This book is not expected to contain assembler code or pseudo code examples.

Any code examples are shown in a `monospace` font.

Feedback

Arm welcomes feedback on its documentation.

Feedback on this book

If you have comments on the content of this book, send an e-mail to arm.psa-feedback@arm.com. Give:

- The title (PSA Cryptography API).
- The number and issue (IHI 0086 1.0.1).
- The page numbers to which your comments apply.
- The rule identifiers to which your comments apply, if applicable.
- A concise explanation of your comments.

Arm also welcomes general suggestions for additions and improvements.

1 Introduction

Arm's Platform Security Architecture (PSA) is a holistic set of threat models, security analyses, hardware and firmware architecture specifications, an open source firmware reference implementation, and an independent evaluation and certification scheme. 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 PSA component that provides a portable interface to cryptographic operations on a wide range of hardware. The interface is user-friendly, while still providing access to the low-level primitives used in modern cryptography. It does not require that the user has access to the key material. Instead, it uses opaque key identifiers.

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

2 Design goals

2.1 Suitable for constrained devices

The interface is suitable for a vast range of devices: from special-purpose cryptographic processors that process data with a built-in key, to constrained devices running custom application code, such as microcontrollers, and multi-application devices, such as servers. Consequentially, the interface is scalable and modular.

- *Scalable*: devices only need to implement the functionality that they will use.
- *Modular*: larger devices implement larger subsets of the same interface, rather than different interfaces.

In this interface, all operations on unbounded amounts of data allow *multi-part* processing, as long as 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 one time. As a result, this specification is suitable for very constrained devices, including those where memory is very limited.

Memory outside the keystore boundary is managed by the application. An implementation of the interface is not required to retain any state between function calls, apart from the content of the keystore and other data that must be kept 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.

2.2 A keystore interface

The specification allows cryptographic operations to be performed on a key to which the application does not have direct access. Except where required for interchange, applications access all keys indirectly, by an identifier. The key material corresponding to that identifier can reside inside a security boundary that prevents it from being extracted, except as permitted by a policy that is defined when the key is created.

2.3 Optional isolation

Implementations can isolate the cryptoprocessor from the calling application, and can further isolate multiple calling applications. The interface allows the implementation to be separated between a frontend and a backend. In an isolated implementation, the frontend is the part of the implementation that is located in the same isolation boundary as the application, which the application accesses by function calls. The backend is the part of the implementation that is located in a different environment, which is protected from the frontend. Various technologies can provide protection, for example:

- Process isolation in an operating system.
- Partition isolation, either with a virtual machine or a partition manager.
- Physical separation between devices.

Communication between the frontend and backend is beyond the scope of this specification.

In an isolated implementation, the backend can 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. The mechanism for identifying callers is beyond the scope of this specification. An implementation that provides caller isolation must document the identification mechanism. An implementation that provides isolation must document any implementation-specific extension of the API that enables frontend instances to share data in any form.

In summary, there are three types of implementation:

- No isolation: there is no security boundary between the application and the cryptoprocessor. For example, a statically or dynamically linked library is an implementation with no isolation.
- Cryptoprocessor isolation: there is a security boundary between the application and the cryptoprocessor, but the cryptoprocessor does not communicate with other applications. For example, a cryptoprocessor chip that is a companion to an application processor is an implementation with cryptoprocessor isolation.
- 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. For example, a cryptography service in a multiprocess environment is an implementation with caller and cryptoprocessor isolation.

2.4 Choice of algorithms

The specification defines a low-level cryptographic interface, where the caller explicitly chooses which algorithm and which security parameters they use. This is necessary to implement protocols that are inescapable in various use cases. The design of the interface enables applications to implement widely-used protocols and data exchange formats, as well as custom ones.

As a consequence, all cryptographic functionality operates according to the precise algorithm specified by the caller. However, this does not apply to device-internal functionality, which does not involve any form of interoperability, such as random number generation. The specification does not include generic higher-level interfaces, where the implementation chooses the best algorithm for a purpose. However, higher-level libraries can be built on top of the PSA Crypto API.

Another consequence is that the specification permits the use of algorithms, key sizes and other parameters that, while known to be insecure, might be necessary to support legacy protocols or legacy data. Where major weaknesses are known, the algorithm descriptions give applicable warnings. However, the lack of a warning both does not and cannot indicate that an algorithm is secure in all circumstances. Application developers need to research the security of the protocols and algorithms that they plan to use to determine if these meet their requirements.

The interface facilitates 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, which takes a parameter that identifies the specific hash algorithm.

2.5 Ease of use

The interface is designed to be as user-friendly as possible, given the aforementioned constraints on suitability for various types of devices and on the freedom to choose algorithms.

In particular, the code flows are designed to reduce the risk of dangerous misuse. The interface is designed in part to make it harder to misuse. Where possible, it is designed so that typical mistakes result in test failures, rather than subtle security issues. Implementations avoid leaking data when a function is called with invalid parameters, to the extent allowed by the C language and by implementation size constraints.

2.6 Example use cases

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

2.6.1 Network Security (TLS)

The API provides all of the cryptographic primitives needed to establish TLS connections.

2.6.2 Secure Storage

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

2.6.3 Network Credentials

The API provides network credential management inside a key store, for example, for X.509-based authentication or pre-shared keys on enterprise networks.

2.6.4 Device Pairing

The API provides support for key agreement protocols that are often used for secure pairing of devices over wireless channels. For example, the pairing of an NFC token or a Bluetooth device might use key agreement protocols upon first use.

2.6.5 Secure Boot

The API provides primitives for use during firmware integrity and authenticity validation, during a secure or trusted boot process.

2.6.6 Attestation

The API provides primitives used in attestation activities. Attestation is the ability for a device to sign an array of bytes with a device private key and return the result to the caller. There are several use cases; ranging 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.

2.6.7 Factory Provisioning

Most IoT devices receive a unique identity during the factory provisioning process, or once they have been deployed to the field. This API provides the APIs necessary for populating a device with keys that represent that identity.

3 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.

Future additions describes features that might be included in future versions of this specification.

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

3.1 Library management

Applications must call `psa_crypto_init()` to initialize the library before using any other function.

3.2 Key management

Applications always access keys indirectly via an identifier, and can perform operations using a key without accessing the key material. This allows keys to be *non-extractable*, where an application can use a key but is not permitted to obtain 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 set of attributes that describe the key and the policy for using the key. A `psa_key_attributes_t` object contains all of the attributes, which is used when creating a key and when querying key attributes.

The key attributes include:

- A type and size that describe the key material.
- The key identifier that the application uses to refer to the key.
- A lifetime that determines when the key material is destroyed, and where it is stored.
- A policy that determines how the key can be used.

Keys are created using one of the *key creation functions*:

- `psa_import_key()`
- `psa_generate_key()`
- `psa_key_derivation_output_key()`
- `psa_copy_key()`

These output the key identifier, that is used to access the key in all other parts of the API.

All of the key attributes are set when the key is created and cannot be changed without destroying the key first. If the original key permits copying, then the application can specify a different lifetime or restricted policy for the copy of the key.

A call to `psa_destroy_key()` destroys the key material, and will cause any active operations that are using the key to fail. Therefore an application must not destroy a key while an operation using that key is in progress, unless the application is prepared to handle a failure of the operation.

3.2.1 Key identifiers

Key identifiers are integral values that act as permanent names for persistent keys, or as transient references to volatile keys. Key identifiers are defined by the application for persistent keys, and by the implementation for volatile keys and for built-in keys.

Key identifiers are output from a successful call to one of the key creation functions.

Valid key identifiers must have distinct values within the same application. If the implementation provides caller isolation, then key identifiers are local to each application. That is, the same key identifier in two applications corresponds to two different keys.

See *Key identifiers* on page 75.

3.2.2 Key lifetimes

The lifetime of a key indicates where it is stored and which application and system actions will create and destroy it.

There are two main types of lifetimes: *volatile* and *persistent*.

Volatile keys are automatically destroyed when the application instance terminates or on a power reset of the device. Volatile key identifiers are allocated by the implementation when the key is created. Volatile keys can be explicitly destroyed with a call to `psa_destroy_key()`.

Persistent keys are preserved until the application explicitly destroys them or until an implementation-specific device management event occurs, for example, a factory reset. The key identifier for a persistent key is set by the application when creating the key, and remains valid throughout the lifetime of the key, even if the application instance that created the key terminates.

See *Key lifetimes* on page 68.

3.2.3 Key policies

All keys have an associated policy that regulates which operations are permitted on the key. Each key policy is a set of usage flags and a specific algorithm that is permitted with the key. See *Key policies* on page 78.

3.2.4 Recommendations of minimum standards for key management

Most implementations provide the following functions:

- `psa_import_key()`. The 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.
- `psa_get_key_attributes()` and the `psa_get_key_xxx()` accessor functions. They are easy to implement, and it is difficult to write applications and to diagnose issues without being able to check the metadata.
- `psa_export_public_key()`. This function is usually provided if the implementation supports any asymmetric algorithm, since public-key cryptography often requires the delivery of a public key that is associated with a protected private key.
- `psa_export_key()`. However, highly constrained implementations that are designed to work only with short-term keys, or only with long-term non-extractable keys, do not need to provide this function.

3.3 Symmetric cryptography

This specification defines interfaces for the following types of symmetric cryptographic operation:

- Message digests, commonly known as hash functions.
- Message authentication codes (MAC).
- Symmetric ciphers.
- Authenticated encryption with associated data (AEAD).

For each type of symmetric cryptographic operation, the API includes:

- A pair of *single-part* functions. For example, compute and verify, or encrypt and decrypt.
- A series of functions that permit *multi-part operations*.

3.3.1 Single-part Functions

Single-part functions are APIs that implement the cryptographic operation in a single function call. This is the easiest API to use when all of the inputs and outputs fit into the application memory.

Some use cases involve messages that are too large to be assembled in memory, or require non-default configuration of the algorithm. These use cases require the use of a *multi-part operation*.

3.3.2 Multi-part operations

Multi-part operations are APIs which split a single cryptographic operation into a sequence of separate steps. This enables fine control over the configuration of the cryptographic operation, and allows the message data to be processed in fragments instead of all at once. For example, the following situations require the use of a multi-part operation:

- Processing messages that cannot be assembled in memory.
- Using a deterministic IV for unauthenticated encryption.
- Providing the IV separately for unauthenticated encryption or decryption.
- Separating the AEAD authentication tag from the cipher text.

Each multi-part operation defines a specific object type to maintain the state of the operation. These types are implementation-defined. All multi-part operations follow the same pattern of use:

1. **Allocate:** Allocate memory for an operation object of the appropriate type. The application can use any allocation strategy: stack, heap, static, etc.
2. **Initialize:** Initialize or assign the operation object by one of the following methods:
   - Set it to logical zero. This is automatic for static and global variables. Explicit initialization must use the associated `PSA_xxx_INIT` macro as the type is implementation-defined.
   - Set it to all-bits zero. This is automatic if the object was allocated with `calloc()`.
   - Assign the value of the associated macro `PSA_xxx_INIT`.
   - Assign the result of calling the associated function `psa_xxx_init()`.

   The resulting object is now *inactive*.

   It is an error to initialize an operation object that is in *active* or *error* states. This can leak memory or other resources.
3. **Setup:** Start a new multi-part operation on an *inactive* operation object. Each operation object will define one or more setup functions to start a specific operation.

   On success, a setup function will put an operation object into an *active* state. On failure, the operation object will remain *inactive*.
4. **Update:** Update an *active* operation object. The update function can provide additional parameters, supply data for processing or generate outputs.

On success, the operation object remains *active*. On failure, the operation object will enter an *error* state.

5. **Finish:** To end the operation, call the applicable finishing function. This will take any final inputs, produce any final outputs, and then release any resources associated with the operation.

   On success, the operation object returns to the *inactive* state. On failure, the operation object will enter an *error* state.

An operation can be aborted at any stage during its use by calling the associated `psa_xxx_abort()` function. This will release any resources associated with the operation and return the operation object to the *inactive* state.

Any error that occurs to an operation while it is in an *active* state will result in the operation entering an *error* state. The application must call the associated `psa_xxx_abort()` function to release the operation resources and return the object to the *inactive* state.

Once an operation object is returned to the *inactive* state, it can be reused by calling one of the applicable setup functions again.

If a multi-part operation object is not initialized before use, the behavior is undefined.

If a multi-part operation function determines that the operation object is not in any valid state, it can return `PSA_ERROR_CORRUPTION_DETECTED`.

If a multi-part operation function is called with an operation object in the wrong state, the function will return `PSA_ERROR_BAD_STATE` and the operation object will enter the *error* state.

It is safe to move a multi-part operation object to a different memory location, for example, using a bitwise copy, and then to use the object in the new location. For example, an application can allocate an operation object on the stack and return it, or the operation object can be allocated within memory managed by a garbage collector. However, this does not permit the following behaviors:

- Moving the object while a function is being called on the object. This is not safe. See also Concurrent calls.
- Working with both the original and the copied operation objects. This requires cloning the operation, which is only available for hash operations using `psa_hash_clone()`.

Each type of multi-part operation can have multiple *active* states. Documentation for the specific operation describes the configuration and update functions, and any requirements about their usage and ordering.

3.3.3 Message digests (Hashes)

The single-part hash functions are:

- `psa_hash_compute()` to calculate the hash of a message.
- `psa_hash_compare()` to compare the hash of a message with a reference value.

The `psa_hash_operation_t` multi-part operation allows messages to be processed in fragments:

1. Initialize the `psa_hash_operation_t` object to zero, or by assigning the value of the associated macro `PSA_HASH_OPERATION_INIT`.

2. Call psa_hash_setup() to specify the required hash algorithm, call psa_hash_clone() to duplicate the state of *active* psa_hash_operation_t object, or call psa_hash_resume() to restart a hash operation with the output from a previously suspended hash operation.
3. Call the psa_hash_update() function on successive chunks of the message.
4. At the end of the message, call the required finishing function:
    - To suspend the hash operation and extract a hash suspend state, call psa_hash_suspend(). The output state can subsequently be used to resume the hash operation.
    - To calculate the digest of a message, call psa_hash_finish().
    - To verify the digest of a message against a reference value, call psa_hash_verify().

To abort the operation or recover from an error, call psa_hash_abort().

3.3.4 Message authentication codes (MACs)

The single-part MAC functions are:

- psa_mac_compute() to calculate the MAC of a message.
- psa_mac_verify() to compare the MAC of a message with a reference value.

The psa_mac_operation_t multi-part operation allows messages to be processed in fragments:

1. Initialize the psa_mac_operation_t object to zero, or by assigning the value of the associated macro PSA_MAC_OPERATION_INIT.
2. Call psa_mac_sign_setup() or psa_mac_verify_setup() to specify the algorithm and key.
3. Call the psa_mac_update() function on successive chunks of the message.
4. At the end of the message, call the required finishing function:
    - To calculate the MAC of the message, call psa_mac_sign_finish().
    - To verify the MAC of the message against a reference value, call psa_mac_verify_finish().

To abort the operation or recover from an error, call psa_mac_abort().

3.3.5 Encryption and decryption

Note:

The unauthenticated cipher API is provided to implement legacy protocols and for use cases where the data integrity and authenticity is guaranteed by non-cryptographic means. It is recommended that newer protocols use *Authenticated encryption (AEAD)* on page 26.

The single-part functions for encrypting or decrypting a message using an unauthenticated symmetric cipher are:

- psa_cipher_encrypt() to encrypt a message using an unauthenticated symmetric cipher. The encryption function generates a random IV. Use the multi-part API to provide a deterministic IV: this is not secure in general, but can be secure in some conditions that depend on the algorithm.

- `psa_cipher_decrypt()` to decrypt a message using an unauthenticated symmetric cipher.

The `psa_cipher_operation_t` multi-part operation permits alternative initialization parameters and allows messages to be processed in fragments:

1. Initialize the `psa_cipher_operation_t` object to zero, or by assigning the value of the associated macro `PSA_CIPHER_OPERATION_INIT`.
2. Call `psa_cipher_encrypt_setup()` or `psa_cipher_decrypt_setup()` to specify the algorithm and key.
3. Provide additional parameters:
   - When encrypting data, generate or set an initialization vector (IV), nonce, or similar initial value such as an initial counter value. 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()`.
   - When decrypting, set the IV or nonce. To set the IV, call `psa_cipher_set_iv()`.
4. Call the `psa_cipher_update()` function on successive chunks of the message.
5. Call `psa_cipher_finish()` to complete the operation and return any final output.

To abort the operation or recover from an error, call `psa_cipher_abort()`.

3.3.6 Authenticated encryption (AEAD)

The single-part AEAD functions are:

- `psa_aead_encrypt()` to encrypt a message using an authenticated symmetric cipher.
- `psa_aead_decrypt()` to decrypt a message using an authenticated symmetric cipher.

These functions follow the interface recommended by *An Interface and Algorithms for Authenticated Encryption* [RFC5116].

The encryption function requires a nonce to be provided. To generate a random nonce, either call `psa_generate_random()` or use the AEAD multi-part API.

The `psa_aead_operation_t` multi-part operation permits alternative initialization parameters and allows messages to be processed in fragments:

1. Initialize the `psa_aead_operation_t` object to zero, or by assigning the value of the associated macro `PSA_AEAD_OPERATION_INIT`.
2. Call `psa_aead_encrypt_setup()` or `psa_aead_decrypt_setup()` to specify the algorithm and key.
3. Provide additional parameters:
   - If the algorithm requires it, call `psa_aead_set_lengths()` to specify the length of the non-encrypted and encrypted inputs to the operation.
   - When encrypting, call either `psa_aead_generate_nonce()` or `psa_aead_set_nonce()` to generate or set the nonce.
   - When decrypting, call `psa_aead_set_nonce()` to set the nonce.
4. Call `psa_aead_update_ad()` zero or more times with fragments of the non-encrypted additional data.
5. Call `psa_aead_update()` zero or more times with fragments of the plaintext or ciphertext to encrypt or decrypt.

6. At the end of the message, call the required finishing function:
   - To complete an encryption operation, call `psa_aead_finish()` to compute and return authentication tag.
   - To complete a decryption operation, call `psa_aead_verify()` to compute the authentication tag and verify it against a reference value.

To abort the operation or recover from an error, call `psa_aead_abort()`.

Having a multi-part interface to authenticated encryption raises specific issues.

Multi-part 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, either 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 multi-part 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.

3.3.7 Key derivation

A key derivation encodes a deterministic method to generate a finite stream of bytes. This data stream is computed by the cryptoprocessor and extracted in chunks. If two key derivation operations are constructed with the same parameters, then they produce the same output.

A key derivation consists of two phases:

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 specification defines a multi-part operation API for key derivation that allows for multiple key and non-key outputs to be extracted from a single derivation operation object.

In an implementation with isolation, the intermediate state of the key derivation is not visible to the caller, and if an output of the derivation is a non-exportable key, then this key cannot be recovered outside the isolation boundary.

Applications use the `psa_key_derivation_operation_t` type to create key derivation operations. The operation object is used as follows:

1. Initialize a `psa_key_derivation_operation_t` object to zero or to `PSA_KEY_DERIVATION_OPERATION_INIT`.
2. Call `psa_key_derivation_setup()` to select a key derivation algorithm.
3. Call the functions `psa_key_derivation_input_bytes()` and `psa_key_derivation_input_key()`, or `psa_key_derivation_key_agreement()` to provide the inputs to the key derivation algorithm. Many key derivation algorithms take multiple inputs; the `step` parameter to these functions indicates which input is being provided. The documentation for each key derivation algorithm describes the expected inputs for that algorithm and in what order to pass them.

4. Optionally, call psa_key_derivation_set_capacity() to set a limit on the amount of data that can be output from the key derivation operation.
5. Call psa_key_derivation_output_key() to create a derived key, or psa_key_derivation_output_bytes() to export the derived data. These functions can be called multiple times to read successive output from the key derivation, until the stream is exhausted when its capacity has been reached.
6. Key derivation does not finish in the same way as other multi-part operations. Call psa_key_derivation_abort() to release the key derivation operation memory when the object is no longer required.

To recover from an error, call psa_key_derivation_abort() to release the key derivation operation memory.

A key derivation operation cannot be rewound. Once a part of the stream has been output, it cannot be output again. This ensures that the same part of the output will not be used for different purposes.

3.3.8 Example of the symmetric cryptography API

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. Derive the message encryption material from the master key.
    a. Initialize a psa_key_derivation_operation_t object to zero or to PSA_KEY_DERIVATION_OPERATION_INIT.
    b. Call psa_key_derivation_setup() with PSA_ALG_HKDF as the algorithm.
    c. Call psa_key_derivation_input_key() with the step PSA_KEY_DERIVATION_INPUT_SECRET and the master key.
    d. Call psa_key_derivation_input_bytes() with the step PSA_KEY_DERIVATION_INPUT_INFO and a public value that uniquely identifies the message.
    e. Populate a psa_key_attributes_t object with the derived message encryption key's attributes.
    f. Call psa_key_derivation_output_key() to create the derived message key.
    g. Call psa_key_derivation_output_bytes() to generate the derived IV.
    h. Call psa_key_derivation_abort() to release the key derivation operation memory.
2. Encrypt the message with the derived material.
    a. Initialize a psa_cipher_operation_t object to zero or to PSA_CIPHER_OPERATION_INIT.
    b. Call psa_cipher_encrypt_setup() with the derived message encryption key.
    c. Call psa_cipher_set_iv() using the derived IV retrieved above.
    d. Call psa_cipher_update() one or more times to encrypt the message.
    e. Call psa_cipher_finish() at the end of the message.
3. Call psa_destroy_key() to clear the generated key.

3.4 Asymmetric cryptography

This specification defines functions for asymmetric cryptography, including asymmetric encryption, asymmetric signature, and two-way key agreement.

3.4.1 Asymmetric encryption

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

3.4.2 Hash-and-sign

The signature and verification functions `psa_sign_message()` and `psa_verify_message()` take a message as one of their inputs and perform a hash-and-sign algorithm.

The functions `psa_sign_hash()` and `psa_verify_hash()` take a message hash as one of their inputs. This is useful for signing pre-computed hashes, or for implementing hash-and-sign using a multi-part hash operation before signing the resulting hash. To determine which hash algorithm to use, call the macro `PSA_ALG_GET_HASH()` on the corresponding signature algorithm.

Some hash-and-sign algorithms add padding to the message hash before completing the signing operation. The format of the padding that is used depends on the algorithm used to construct the signature.

3.4.3 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 approach is to use a key derivation operation with the `psa_key_derivation_key_agreement()` input function, which calculates a shared secret for the key derivation function.

Where an application needs direct access to the shared secret, it can call `psa_raw_key_agreement()` instead. Note that in general the shared secret is not directly suitable for use as a key because it is biased.

3.5 Randomness and key generation

We 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 can 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 can omit this.

It is recommended that applications use `psa_generate_key()`, `psa_cipher_generate_iv()` or `psa_aead_generate_nonce()` 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.

4 Sample architectures

This section describes some example architectures that can be used for implementations of the interface described in this specification. This list is not exhaustive and the section is entirely non-normative.

4.1 Single-partition architecture

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

This architecture does not conform to the Arm *Platform Security Architecture Security Model*. However, it is useful for providing cryptographic services that use the same interface, even on devices that cannot support any security boundary. So, while this architecture is not the primary design goal of the API defined in the present specification, it is supported.

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, though it shares this high-security partition with other platform security services.

4.2 Cryptographic token and single-application processor

This system is composed of two partitions: one 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, the architecture provides cryptoprocessor isolation. The cryptoprocessor has some non-volatile storage, a TRNG, and possibly, some cryptographic accelerators.

There are a number of potential physical realizations: the cryptoprocessor might 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.

4.3 Cryptoprocessor with no key storage

As in the *Cryptographic token and single-application processor* architecture, this system is also composed of two partitions separated by a security boundary and also provides cryptoprocessor isolation. However, unlike the previous architecture, in this system, 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 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 might be a hardware unique key that is burnt to one-time programmable memory during the manufacturing of the device. This key can be used for one or more purposes:

- Encrypt and authenticate data stored in the application processor.
- Communicate with a paired device.

- Allow the application to perform operations with keys that are derived from the hardware unique key.

4.4 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.

4.5 Multi-cryptoprocessor architecture

This system includes multiple cryptoprocessors. There are several reasons to have multiple cryptoprocessors:

- Different compromises between security and performance for different keys. Typically, this means a cryptoprocessor that runs on the same hardware as the main application and processes short-term secrets, a secure element or a similar separate chip that retains long-term secrets.
- Independent provisioning of certain secrets.
- A combination of a non-removable cryptoprocessor and removable ones, for example, a smartcard or HSM.
- Cryptoprocessors managed by different stakeholders who do not trust each other.

The keystore implementation needs to dispatch each request to the correct processor. For example:

- All requests involving a non-extractable key must be processed in the cryptoprocessor that holds that key.
- Requests involving a persistent key must be processed in the cryptoprocessor that corresponds to the key's lifetime value.
- Requests involving a volatile key might target a cryptoprocessor based on parameters supplied by the application, or based on considerations such as performance inside the implementation.

5 Library conventions

5.1 Error handling

5.1.1 Return status

Almost all functions return a status indication of type `psa_status_t`. This is an enumeration of integer values, with `0` (`PSA_SUCCESS`) indicating successful operation and other values indicating errors. The exceptions are functions which only access objects that are intended to be implemented as simple data structures. Such functions cannot fail and either return `void` or a data value.

Unless specified otherwise, if multiple error conditions apply, an implementation is free to return any of the applicable error codes. The choice of error code is considered an implementation quality issue. Different implementations can make different choices, for example to favor code size over ease of debugging or vice versa.

If the behavior is undefined, for example, if a function receives an invalid pointer as a parameter, this specification makes no guarantee that the function will return an error. Implementations are encouraged to return an error or halt the application in a manner that is appropriate for the platform if the undefined behavior condition can be detected. However, application developers need to be aware that undefined behavior conditions cannot be detected in general.

5.1.2 Behavior on error

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 the output parameters is undefined, but otherwise the state of the system has not changed, except as described below.

In general, functions that modify the system state, for example, creating or destroying a key, must leave the system state unchanged if they return an error code. There are specific conditions that can result in different behavior:

- The status `PSA_ERROR_BAD_STATE` indicates that a parameter was not in a valid state for the requested action. This parameter might have been modified by the call and is now in an undefined state. The only valid action on an object in an undefined state is to abort it with the appropriate `psa_abort_xxx()` function.
- The status `PSA_ERROR_INSUFFICIENT_DATA` indicates that a key derivation object has reached its maximum capacity. The key derivation operation might have been modified by the call. Any further attempt to obtain output from the key derivation operation will return `PSA_ERROR_INSUFFICIENT_DATA`.
- The status `PSA_ERROR_COMMUNICATION_FAILURE` indicates that the communication between the application and the cryptoprocessor has broken down. In this case, the cryptoprocessor must either finish the requested action successfully, or interrupt the action and roll back the system to its original state. Because it is often impossible to report the outcome to the application after a communication failure, this specification does not provide a way for the application to determine whether the action was successful.
- The statuses `PSA_ERROR_STORAGE_FAILURE`, `PSA_ERROR_DATA_CORRUPT`, `PSA_ERROR_HARDWARE_FAILURE` and `PSA_ERROR_CORRUPTION_DETECTED` might indicate data corruption in the system state. When a function returns one of these statuses, the system state might have changed from its previous state before the function call, even though the function call failed.
- Some system states cannot be rolled back, for example, the internal state of the random number generator or the content of access logs.

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

5.2 Parameter conventions

5.2.1 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. Passing a null pointer as a function parameter in other cases is expected to abort the caller on implementations where this is the normal behavior for a null pointer dereference.

Pointers to input parameters can 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 *Stability of parameters* on page 34 section.

5.2.2 Input buffer sizes

For input buffers, the parameter convention is:

`const uint8_t *foo`
: Pointer to the first byte of the data. The pointer can 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.

5.2.3 Output buffer sizes

For output buffers, the parameter convention is:

`uint8_t *foo`
: Pointer to the first byte of the data. The pointer can 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 errors is unspecified, unless explicitly mentioned in the function description. They might 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 an individual function's 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 can be invalid if the buffer size is `0`.

`size_t foo_length`
: The number of bytes to return in `foo` if successful.

5.2.4 Overlap between parameters

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

Output buffers can overlap with input buffers. In this event, the implementation must return the same result as if the buffers did not overlap. The implementation must behave as if it had copied all the inputs into temporary memory, as far as the result is concerned. However, it is possible that overlap between parameters will affect the performance of a function call. Overlap might also affect memory management security if the buffer is located in memory that the caller shares with another security context, as described in the *Stability of parameters* section.

5.2.5 Stability of parameters

In some environments, it is possible for the content of a parameter to change while a function is executing. It might 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, 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 the assumptions that an implementation can make about function parameters, and the guarantees that the implementation must provide about how it accesses parameters.

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 must not be modified during the execution, and the value of an output parameter is undetermined until the function returns. The implementation can read an input parameter that is not a buffer multiple times and expect to read the same data. The implementation can 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 carefully accesses non-overlapping buffer parameters in order to prevent any security risk resulting from the content of the buffer being modified or observed during the execution of the function. In an input buffer that does not overlap with an output buffer, the implementation reads each byte of the input once, at most. The implementation does not read from an output buffer that does not overlap with an input buffer. Additionally, the implementation does 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.

Unless otherwise specified, the implementation must not keep a reference to any parameter once a function call has returned.

5.3 Key types and algorithms

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

There is some overlap in the information conveyed by key types and algorithms. Both types contain 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 might 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`.

5.3.1 Structure of key and algorithm types

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

The encodings follows a few conventions:

- The highest bit is a vendor flag. Current and future versions of this specification will only define values where this bit is clear. Implementations that wish to define additional implementation-specific values must use values where this bit is set, to avoid conflicts with future versions of this specification.
- The next few highest bits indicate the corresponding algorithm category: hash, MAC, symmetric cipher, asymmetric encryption, and so on.
- 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.

5.4 Concurrent calls

In some environments, an application can make calls to the PSA crypto API in separate threads. In such an environment, *concurrent calls* are two or more calls to the API whose execution can overlap in time.

Concurrent calls are performed correctly, as if the calls were executed in sequence, provided that they obey the following constraints:

- There is no overlap between an output parameter of one call and an input or output parameter of another call. Overlap between input parameters is permitted.

- A call to destroy a key must not overlap with a concurrent call to any of the following functions:
  - Any call where the same key identifier is a parameter to the call.
  - Any call in a multi-part operation, where the same key identifier was used as a parameter to a previous step in the multi-part operation.
- Concurrent calls must not use the same operation object.

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

If the application modifies an input parameter while a function call is in progress, the behavior is undefined.

Individual implementations can provide additional guarantees.

6 Implementation considerations

6.1 Implementation-specific aspects of the interface

6.1.1 Implementation profile

Implementations can 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 for each implementation must describe the profile that it implements. This specification's companion documents also define a number of standard profiles.

6.1.2 Implementation-specific types

This specification defines a number of implementation-specific types, which represent objects whose content depends on the implementation. These are defined as C `typedef` types in this specification, with a comment */* implementation-defined type */* in place of the underlying type definition. For some types the specification constrains the type, for example, by requiring that the type is a `struct`, or that it is convertible to and from an unsigned integer. In the implementation's version of `psa/crypto.h`, these types need to be defined as complete C types so that objects of these types can be instantiated by application code.

Applications that rely on the implementation specific definition of any of these types might not be portable to other implementations of this specification.

6.1.3 Implementation-specific macros

Some macro constants and function-like macros are precisely defined by this specification. The use of an exact definition is essential if the definition can appear in more than one header file within a compilation.

Other macros that are defined by this specification have a macro body that is implementation-specific. The description of an implementation-specific macro can optionally specify each of the following requirements:

- Input domains: the macro must be valid for arguments within the input domain.
- A return type: the macro result must be compatible with this type.
- Output range: the macro result must lie in the output range.
- Computed value: A precise mapping of valid input to output values.

Each implementation-specific macro is in one of following categories:

Specification-defined value

The result type and computed value of the macro expression is defined by this specification, but the definition of the macro body is provided by the implementation.

These macros are indicated in this specification using the comment */* specification-defined value */*.

For function-like macros with specification-defined values:

- Example implementations are provided in an appendix to this specification. See *Example macro implementations* on page 237.
- The expected computation for valid and supported input arguments will be defined as pseudo-code in a future version of this specification.

Implementation-defined value

The value of the macro expression is implementation-defined.

For some macros, the computed value is derived from the specification of one or more cryptographic algorithms. In these cases, the result must exactly match the value in those external specifications.

These macros are indicated in this specification using the comment */* implementation-defined value */*.

Some of these macros compute a result based on an algorithm or key type. If an implementation defines vendor-specific algorithms or key types, then it must provide an implementation for such macros that takes all relevant algorithms and types into account. Conversely, an implementation that does not support a certain algorithm or key type can 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, an implementation is allowed to require a buffer size that is larger than the theoretical minimum. An implementation must define minimum-size macros in such a way that it guarantees that the buffer of the resulting size is sufficient for the output of the corresponding function. Refer to each macro's documentation for the applicable requirements.

6.2 Porting to a platform

6.2.1 Platform assumptions

This specification is designed for a C99 platform. The interface is defined in terms of C macros, functions and objects.

The specification assumes 8-bit bytes, and "byte" and "octet" are used synonymously.

6.2.2 Platform-specific types

The specification makes use of some types defined in C99. These types must be defined in the implementation version of `psa/crypto.h` or by a header included in this file. The following C99 types are used:

`uint8_t`, `uint16_t`, `uint32_t`
: Unsigned integer types with 8, 16 and 32 value bits respectively. These types are defined by the C99 header `stdint.h`.

6.2.3 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 that calls drivers for hardware accelerators and external cryptographic hardware.

6.3 Security requirements and recommendations

6.3.1 Error detection

Implementations that provide isolation between the caller and the cryptography processing environment must 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 are recommended to detect bad parameters and fail-safe where possible.

6.3.2 Indirect object references

Implementations can use different strategies for allocating key identifiers, and other types of indirect object reference.

Implementations that provide isolation between the caller and the cryptography processing environment must consider the threats relating to abuse and misuse of key identifiers and other indirect resource references. For example, multi-part operations can be implemented as backend state to which the client only maintains an indirect reference in the application's multi-part operation object.

An implementation that supports multiple callers must implement strict isolation of API resources between different callers. For example, a client must not be able to obtain a reference to another client's key by guessing the key identifier value. Isolation of key identifiers can be achieved in several ways. For example:

- There is a single identifier namespace for all clients, and the implementation verifies that the client is the owner of the identifier when looking up the key.
- Each client has an independent identifier namespace, and the implementation uses a client specific identifier-to-key mapping when looking up the key.

After a volatile key identifier is destroyed, it is recommended that the implementation does not immediately reuse the same identifier value for a different key. This reduces the risk of an attack that is able to exploit a key identifier reuse vulnerability within an application.

6.3.3 Memory cleanup

Implementations must wipe all sensitive data from memory when it is no longer used. It is recommended that they wipe this sensitive data as soon as possible. All temporary data used during the execution of a function, such as stack buffers, must be wiped before the function returns. All data associated with an object, such as a multi-part operation, must be wiped, at the latest, when the object becomes inactive, for example, when a multi-part operation is aborted.

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

6.3.4 Managing key material

In implementations that have limited volatile memory for keys, the implementation is permitted to store a volatile key to a temporary location in non-volatile memory. The implementation must delete any such copies when the key is destroyed, and it is recommended that these copies are deleted as soon as the key is reloaded into volatile memory. An implementation that uses this method must clear any stored volatile key material on startup.

Implementing the memory cleanup rule for persistent keys can result in inefficiencies when the same persistent key is used sequentially in multiple cryptographic operations. The inefficiency stems from loading the key from non-volatile storage on each use of the key. The `PSA_KEY_USAGE_CACHE` usage flag in a key policy allows an application to request that the implementation does not cleanup non-essential copies of persistent key material, effectively suspending the cleanup rules for that key. The effects of this policy depend on the implementation and the key, for example:

- For volatile keys or keys in a secure element with no open/close mechanism, this is likely to have no effect.
- For persistent keys that are not in a secure element, this allows the implementation to keep the key in a memory cache outside of the memory used by ongoing operations.
- For keys in a secure element with an open/close mechanism, this is a hint to keep the key open in the secure element.

The application can indicate when it has finished using the key by calling `psa_purge_key()`, to request that the key material is cleaned from memory.

6.3.5 Safe outputs on error

Implementations must ensure that confidential data is not written to output parameters before validating that the disclosure of this confidential data is authorized. This requirement is particularly important for implementations where the caller can share memory with another security context, as described in the Stability of parameters section.

In most cases, the specification does not define the content of output parameters when an error occurs. It is recommended that implementations try to ensure that the content of output parameters is as safe as possible, in case an application flaw or a data leak causes it to be used. In particular, Arm recommends that implementations avoid placing partial output in output buffers when an action is interrupted. The meaning of “safe as possible” depends on the implementation, as different environments require different compromises between implementation complexity, overall robustness and performance. Some common strategies are to leave output parameters unchanged, in case of errors, or zeroing them out.

6.3.6 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. There is a vast body of literature 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 specify particular requirements for attack resistance. Implementers are encouraged to consider the attack resistance desired in each use case and design their implementation accordingly. Security standards for attack resistance for particular targets might be applicable in certain use cases.

6.4 Other implementation considerations

6.4.1 Philosophy of resource management

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

Alternatively, the interface is also designed for conservative strategies of memory management. An implementation can avoid dynamic memory allocation altogether by obeying certain restrictions:

- Pre-allocate memory for a predefined number of keys, each with sufficient memory for all key types that can be stored.
- For multi-part 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.

7 Usage considerations

7.1 Security recommendations

7.1.1 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 the 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 might 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, the accelerator might fail, even when the application processor keeps running normally.

- Most functions might fail due to a lack of resources. However, some implementations guarantee that certain functions always have sufficient memory.
- All functions that access persistent keys might fail due to a storage failure.
- All functions that require randomness might 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.

7.1.2 Shared memory and concurrency

Some environments allow applications to be multithreaded, while others do not. In some environments, applications can share memory with a different security context. In environments with multithreaded applications or shared memory, 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, then the behavior is only well-defined if all the calls are only reading from the object and do not modify it. Read accesses include reading memory by input parameters and reading keystore content by using a key. For more details, refer to the Concurrent calls section.

If an application shares memory with another security context, it can pass shared memory blocks as input buffers or output buffers, but not as non-buffer parameters. For more details, refer to the *Stability of parameters* on page 34 section.

7.1.3 Cleaning up after use

To minimize impact if the system is compromised, it is recommended that applications wipe all sensitive data from memory when it is no longer used. That way, only data that is currently in use can be leaked, and past data is not compromised.

Wiping sensitive data includes:

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

8 Library management reference

8.1 PSA status codes

8.1.1 Status type

`psa_status_t` (type)

Function return status.

```
typedef int32_t psa_status_t;
```

This is either PSA_SUCCESS, which is zero, indicating success; or a small negative value indicating that an error occurred. Errors are encoded as one of the PSA_ERROR_xxx values defined here.

8.1.2 Success codes

PSA_SUCCESS (macro)

The action was completed successfully.

```
#define PSA_SUCCESS ((psa_status_t)0)
```

8.1.3 Error codes

PSA_ERROR_GENERIC_ERROR (macro)

An error occurred that does not correspond to any defined failure cause.

```
#define PSA_ERROR_GENERIC_ERROR ((psa_status_t)-132)
```

Implementations can use this error code if none of the other standard error codes are applicable.

PSA_ERROR_NOT_SUPPORTED (macro)

The requested operation or a parameter is not supported by this implementation.

```
#define PSA_ERROR_NOT_SUPPORTED ((psa_status_t)-134)
```

It is recommended that implementations return this error code when an enumeration parameter such as a key type, algorithm, etc. is not recognized. If a combination of parameters is recognized and identified as not valid, return PSA_ERROR_INVALID_ARGUMENT instead.

PSA_ERROR_NOT_PERMITTED (macro)

The requested action is denied by a policy.

```
#define PSA_ERROR_NOT_PERMITTED ((psa_status_t)-133)
```

It is recommended that implementations return this error code when the parameters are recognized as valid and supported, and a policy explicitly denies the requested operation.

If a subset of the parameters of a function call identify a forbidden operation, and another subset of the parameters are not valid or not supported, it is unspecified whether the function returns PSA_ERROR_NOT_PERMITTED, PSA_ERROR_NOT_SUPPORTED or PSA_ERROR_INVALID_ARGUMENT.

PSA_ERROR_BUFFER_TOO_SMALL (macro)

An output buffer is too small.

```
#define PSA_ERROR_BUFFER_TOO_SMALL ((psa_status_t)-138)
```

Applications can call the `PSA_xxx_SIZE` macro listed in the function description to determine a sufficient buffer size.

It is recommended that implementations only return this error code in cases when performing the operation with a larger output buffer would succeed. However, implementations can also return this error if a function has invalid or unsupported parameters in addition to an insufficient output buffer size.

PSA_ERROR_ALREADY_EXISTS (macro)

Asking for an item that already exists.

```
#define PSA_ERROR_ALREADY_EXISTS ((psa_status_t)-139)
```

It is recommended that implementations return this error code when attempting to write to a location where a key is already present.

PSA_ERROR_DOES_NOT_EXIST (macro)

Asking for an item that doesn't exist.

```
#define PSA_ERROR_DOES_NOT_EXIST ((psa_status_t)-140)
```

Implementations must not return this error code to indicate that a key identifier is invalid, but must return `PSA_ERROR_INVALID_HANDLE` instead.

PSA_ERROR_BAD_STATE (macro)

The requested action cannot be performed in the current state.

```
#define PSA_ERROR_BAD_STATE ((psa_status_t)-137)
```

Multi-part operations return this error when one of the functions is called out of sequence. Refer to the function descriptions for permitted sequencing of functions.

Implementations must not return this error code to indicate that a key identifier is invalid, but must return `PSA_ERROR_INVALID_HANDLE` instead.

PSA_ERROR_INVALID_ARGUMENT (macro)

The parameters passed to the function are invalid.

```
#define PSA_ERROR_INVALID_ARGUMENT ((psa_status_t)-135)
```

Implementations can return this error any time a parameter or combination of parameters are recognized as invalid.

Implementations must not return this error code to indicate that a key identifier is invalid, but must return `PSA_ERROR_INVALID_HANDLE` instead.

PSA_ERROR_INSUFFICIENT_MEMORY (macro)

There is not enough runtime memory.

```
#define PSA_ERROR_INSUFFICIENT_MEMORY ((psa_status_t)-141)
```

If the action is carried out across multiple security realms, this error can refer to available memory in any of the security realms.

PSA_ERROR_INSUFFICIENT_STORAGE (macro)

There is not enough persistent storage.

```
#define PSA_ERROR_INSUFFICIENT_STORAGE ((psa_status_t)-142)
```

Functions that modify the key storage return this error code if there is insufficient storage space on the host media. In addition, many functions that do not otherwise access storage might return this error code if the implementation requires a mandatory log entry for the requested action and the log storage space is full.

PSA_ERROR_COMMUNICATION_FAILURE (macro)

There was a communication failure inside the implementation.

```
#define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)-145)
```

This can indicate a communication failure between the application and an external cryptoprocessor or between the cryptoprocessor and an external volatile or persistent memory. A communication failure can be transient or permanent depending on the cause.

> **Warning:** If a function returns this error, it is undetermined whether the requested action has completed. Returning `PSA_SUCCESS` is recommended on successful completion whenever possible, however functions can return `PSA_ERROR_COMMUNICATION_FAILURE` if the requested action was completed successfully in an external cryptoprocessor but there was a breakdown of communication before the cryptoprocessor could report the status to the application.

PSA_ERROR_STORAGE_FAILURE (macro)

There was a storage failure that might have led to data loss.

```
#define PSA_ERROR_STORAGE_FAILURE ((psa_status_t)-146)
```

This error indicates that some persistent storage could not be read or written by the implementation. It does not indicate the following situations, which have specific error codes:

- A corruption of volatile memory - use `PSA_ERROR_CORRUPTION_DETECTED`.
- A communication error between the cryptoprocessor and its external storage - use `PSA_ERROR_COMMUNICATION_FAILURE`.
- When the storage is in a valid state but is full - use `PSA_ERROR_INSUFFICIENT_STORAGE`.

- When the storage or stored data is corrupted - use PSA_ERROR_DATA_CORRUPT.
- When the stored data is not valid - use PSA_ERROR_DATA_INVALID.

A storage failure does not indicate that any data that was previously read is invalid. However this previously read data might no longer be readable from storage.

When a storage failure occurs, it is no longer possible to ensure the global integrity of the keystore. Depending on the global integrity guarantees offered by the implementation, access to other data might fail even if the data is still readable but its integrity cannot be guaranteed.

It is recommended to only use this error code to report a permanent storage corruption. However application writers must keep in mind that transient errors while reading the storage might be reported using this error code.

PSA_ERROR_DATA_CORRUPT (macro)

Stored data has been corrupted.

```
#define PSA_ERROR_DATA_CORRUPT ((psa_status_t)-152)
```

This error indicates that some persistent storage has suffered corruption. It does not indicate the following situations, which have specific error codes:

- A corruption of volatile memory - use PSA_ERROR_CORRUPTION_DETECTED.
- A communication error between the cryptoprocessor and its external storage - use PSA_ERROR_COMMUNICATION_FAILURE.
- When the storage is in a valid state but is full - use PSA_ERROR_INSUFFICIENT_STORAGE.
- When the storage fails for other reasons - use PSA_ERROR_STORAGE_FAILURE.
- When the stored data is not valid - use PSA_ERROR_DATA_INVALID.

Note that a storage corruption does not indicate that any data that was previously read is invalid. However this previously read data might no longer be readable from storage.

When a storage failure occurs, it is no longer possible to ensure the global integrity of the keystore. Depending on the global integrity guarantees offered by the implementation, access to other data might fail even if the data is still readable but its integrity cannot be guaranteed.

It is recommended to only use this error code to report when a storage component indicates that the stored data is corrupt, or fails an integrity check. For example, in situations that the *PSA Storage API* [PSA-ITS] reports PSA_ERROR_DATA_CORRUPT or PSA_ERROR_INVALID_SIGNATURE.

PSA_ERROR_DATA_INVALID (macro)

Data read from storage is not valid for the implementation.

```
#define PSA_ERROR_DATA_INVALID ((psa_status_t)-153)
```

This error indicates that some data read from storage does not have a valid format. It does not indicate the following situations, which have specific error codes:

- When the storage or stored data is corrupted - use PSA_ERROR_DATA_CORRUPT.

- When the storage fails for other reasons - use PSA_ERROR_STORAGE_FAILURE.
- An invalid argument to the API - use PSA_ERROR_INVALID_ARGUMENT.

This error is typically a result of an integration failure, where the implementation reading the data is not compatible with the implementation that stored the data.

It is recommended to only use this error code to report when data that is successfully read from storage is invalid.

PSA_ERROR_HARDWARE_FAILURE (macro)

A hardware failure was detected.

```
#define PSA_ERROR_HARDWARE_FAILURE ((psa_status_t)-147)
```

A hardware failure can be transient or permanent depending on the cause.

PSA_ERROR_CORRUPTION_DETECTED (macro)

A tampering attempt was detected.

```
#define PSA_ERROR_CORRUPTION_DETECTED ((psa_status_t)-151)
```

If an application receives this error code, there is no guarantee that previously accessed or computed data was correct and remains confidential. In this situation, it is recommended that applications perform no further security functions and enter a safe failure state.

Implementations can return this error code if they detect an invalid state that cannot happen during normal operation and that indicates that the implementation's security guarantees no longer hold. Depending on the implementation architecture and on its security and safety goals, the implementation might forcibly terminate the application.

This error code is intended as a last resort when a security breach is detected and it is unsure whether the keystore data is still protected. Implementations must only return this error code to report an alarm from a tampering detector, to indicate that the confidentiality of stored data can no longer be guaranteed, or to indicate that the integrity of previously returned data is now considered compromised. Implementations must not use this error code to indicate a hardware failure that merely makes it impossible to perform the requested operation, instead use PSA_ERROR_COMMUNICATION_FAILURE, PSA_ERROR_STORAGE_FAILURE, PSA_ERROR_HARDWARE_FAILURE, PSA_ERROR_INSUFFICIENT_ENTROPY or other applicable error code.

This error indicates an attack against the application. Implementations must not return this error code as a consequence of the behavior of the application itself.

PSA_ERROR_INSUFFICIENT_ENTROPY (macro)

There is not enough entropy to generate random data needed for the requested action.

```
#define PSA_ERROR_INSUFFICIENT_ENTROPY ((psa_status_t)-148)
```

This error indicates a failure of a hardware random generator. Application writers must note that this error can be returned not only by functions whose purpose is to generate random data, such as key, IV or nonce generation, but also by functions that execute an algorithm with a randomized result, as well as functions that use randomization of intermediate computations as a countermeasure to certain attacks.

It is recommended that implementations do not return this error after `psa_crypto_init()` has succeeded. This can be achieved if the implementation generates sufficient entropy during initialization and subsequently a cryptographically secure pseudorandom generator (PRNG) is used. However, implementations might return this error at any time, for example, if a policy requires the PRNG to be reseeded during normal operation.

PSA_ERROR_INVALID_SIGNATURE (macro)

The signature, MAC or hash is incorrect.

```
#define PSA_ERROR_INVALID_SIGNATURE ((psa_status_t)-149)
```

Verification functions return this error if the verification calculations completed successfully, and the value to be verified was determined to be incorrect.

If the value to verify has an invalid size, implementations can return either `PSA_ERROR_INVALID_ARGUMENT` or `PSA_ERROR_INVALID_SIGNATURE`.

PSA_ERROR_INVALID_PADDING (macro)

The decrypted padding is incorrect.

```
#define PSA_ERROR_INVALID_PADDING ((psa_status_t)-150)
```

> **Warning:** In some protocols, when decrypting data, it is essential that the behavior of the application does not depend on whether the padding is correct, down to precise timing. Protocols that use authenticated encryption are recommended for use by applications, rather than plain encryption. If the application must perform a decryption of unauthenticated data, the application writer must take care not to reveal whether the padding is invalid.

Implementations must handle padding carefully, aiming to make it impossible for an external observer to distinguish between valid and invalid padding. In particular, it is recommended that the timing of a decryption operation does not depend on the validity of the padding.

PSA_ERROR_INSUFFICIENT_DATA (macro)

Return this error when there's insufficient data when attempting to read from a resource.

```
#define PSA_ERROR_INSUFFICIENT_DATA ((psa_status_t)-143)
```

PSA_ERROR_INVALID_HANDLE (macro)

The key identifier is not valid.

```
#define PSA_ERROR_INVALID_HANDLE ((psa_status_t)-136)
```

See also *Key identifiers* on page 21.

8.2 PSA Crypto library

8.2.1 API version

PSA_CRYPTO_API_VERSION_MAJOR (macro)

The major version of this implementation of the PSA Crypto API.

```
#define PSA_CRYPTO_API_VERSION_MAJOR 1
```

PSA_CRYPTO_API_VERSION_MINOR (macro)

The minor version of this implementation of the PSA Crypto API.

```
#define PSA_CRYPTO_API_VERSION_MINOR 0
```

8.2.2 Library initialization

psa_crypto_init (function)

Library initialization.

```
psa_status_t psa_crypto_init(void);
```

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_INSUFFICIENT_ENTROPY

Description

Applications must call this function before calling any other function in this module.

Applications are permitted to call this function more than once. Once a call succeeds, subsequent calls are guaranteed to succeed.

If the application calls other functions before calling psa_crypto_init(), the behavior is undefined. In this situation:

- Implementations are encouraged to either perform the operation as if the library had been initialized or to return PSA_ERROR_BAD_STATE or some other applicable error.
- Implementations must not return a success status if the lack of initialization might have security implications, for example due to improper seeding of the random number generator.

9 Key management reference

9.1 Key attributes

Key attributes are managed in a `psa_key_attributes_t` object. These are used when a key is created, after which the key attributes are fixed. Attributes of an existing key can be queried using `psa_get_key_attributes()`.

Description of the individual attributes is found in the following sections:

- *Key types* on page 53
- *Key identifiers* on page 75
- *Key lifetimes* on page 68
- *Key policies* on page 78

9.1.1 Managing key attributes

`psa_key_attributes_t` (type)

The type of an object containing key attributes.

```
typedef /* implementation-defined type */ psa_key_attributes_t;
```

This is the object that represents the metadata of a key object. Metadata that can be stored in attributes includes:

- The location of the key in storage, indicated by its key identifier and its lifetime.
- The key's policy, comprising usage flags and a specification of the permitted algorithm(s).
- Information about the key itself: the key type and its size.
- Implementations can define additional attributes.

The actual key material is not considered an attribute of a key. Key attributes do not contain information that is generally considered highly confidential.

> **Note:**
>
> Implementations are recommended to define the attribute object as a simple data structure, with fields corresponding to the individual key attributes. In such an implementation, each function `psa_set_key_xxx()` sets a field and the corresponding function `psa_get_key_xxx()` retrieves the value of the field.
>
> An implementations can report attribute values that are equivalent to the original one, but have a different encoding. For example, an implementation can use a more compact representation for types where many bit-patterns are invalid or not supported, and store all values that it does not support as a special marker value. In such an implementation, after setting an invalid value, the corresponding get function returns an invalid value which might not be the one that was originally stored.

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

An attribute object can contain references to auxiliary resources, for example pointers to allocated memory or indirect references to pre-calculated values. In order to free such resources, the application must call `psa_reset_key_attributes()`. As an exception, calling `psa_reset_key_attributes()` on an attribute object is optional if the object has only been modified by the following functions since it was initialized or last reset with `psa_reset_key_attributes()`:

- `psa_set_key_id()`
- `psa_set_key_lifetime()`
- `psa_set_key_type()`
- `psa_set_key_bits()`
- `psa_set_key_usage_flags()`
- `psa_set_key_algorithm()`

Before calling any function on a key attribute object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

```
psa_key_attributes_t attributes;
memset(&attributes, 0, sizeof(attributes));
```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

```
static psa_key_attributes_t attributes;
```

- Initialize the object to the initializer `PSA_KEY_ATTRIBUTES_INIT`, for example:

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
```

- Assign the result of the function `psa_key_attributes_init()` to the object, for example:

```
psa_key_attributes_t attributes;
attributes = psa_key_attributes_init();
```

A freshly initialized attribute object contains the following values:

Attribute	Value
lifetime	PSA_KEY_LIFETIME_VOLATILE.
key identifier	PSA_KEY_ID_NULL - which is not a valid key identifier.
type	PSA_KEY_TYPE_NONE - meaning that the type is unspecified.
key size	0 - meaning that the size is unspecified.
usage flags	0 - which allows no usage except exporting a public key.
algorithm	PSA_ALG_NONE - which does not allow cryptographic usage, but allows exporting.

Usage

A typical sequence to create a key is as follows:

1. Create and initialize an attribute object.
2. If the key is persistent, call psa_set_key_id(). Also call psa_set_key_lifetime() to place the key in a non-default location.
3. Set the key policy with psa_set_key_usage_flags() and psa_set_key_algorithm().
4. Set the key type with psa_set_key_type(). Skip this step if copying an existing key with psa_copy_key().
5. When generating a random key with psa_generate_key() or deriving a key with psa_key_derivation_output_key(), set the desired key size with psa_set_key_bits().
6. Call a key creation function: psa_import_key(), psa_generate_key(), psa_key_derivation_output_key() or psa_copy_key(). This function reads the attribute object, creates a key with these attributes, and outputs an identifier for the newly created key.
7. Optionally call psa_reset_key_attributes(), now that the attribute object is no longer needed. Currently this call is not required as the attributes defined in this specification do not require additional resources beyond the object itself.

A typical sequence to query a key's attributes is as follows:

1. Call psa_get_key_attributes().
2. Call psa_get_key_xxx() functions to retrieve the required attribute(s).
3. Call psa_reset_key_attributes() to free any resources that can be used by the attribute object.

Once a key has been created, it is impossible to change its attributes.

PSA_KEY_ATTRIBUTES_INIT (macro)

This macro returns a suitable initializer for a key attribute object of type psa_key_attributes_t.

```
#define PSA_KEY_ATTRIBUTES_INIT /* implementation-defined value */
```

`psa_key_attributes_init` (function)

Return an initial value for a key attribute object.

```
psa_key_attributes_t psa_key_attributes_init(void);
```

Returns: `psa_key_attributes_t`

`psa_get_key_attributes` (function)

Retrieve the attributes of a key.

```
psa_status_t psa_get_key_attributes(psa_key_id_t key,
                                    psa_key_attributes_t * attributes);
```

Parameters

`key`	Identifier of the key to query.
`attributes`	On entry, `*attributes` must be in a valid state. On successful return, it contains the attributes of the key. On failure, it is equivalent to a freshly-initialized attribute object.

Returns: `psa_status_t`

`PSA_SUCCESS`	
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function first resets the attribute object as with `psa_reset_key_attributes()`. It then copies the attributes of the given key into the given attribute object.

> **Note:**
>
> This function clears any previous content from the attribute object and therefore expects it to be in a valid state. In particular, if this function is called on a newly allocated attribute object, the attribute object must be initialized before calling this function.

> **Note:**

This function might allocate memory or other resources. Once this function has been called on an attribute object, `psa_reset_key_attributes()` must be called to free these resources.

`psa_reset_key_attributes` (function)

Reset a key attribute object to a freshly initialized state.

```
void psa_reset_key_attributes(psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The attribute object to reset.

Returns: `void`

Description

The attribute object must be initialized as described in the documentation of the type `psa_key_attributes_t` before calling this function. Once the object has been initialized, this function can be called at any time.

This function frees any auxiliary resources that the object might contain.

9.2 Key types

9.2.1 Key type encoding

`psa_key_type_t` (type)

Encoding of a key type.

```
typedef uint16_t psa_key_type_t;
```

This is a structured bitfield that identifies the category and type of key. The range of key type values is divided as follows:

`PSA_KEY_TYPE_NONE == 0`
: Reserved as an invalid key type.

`0x0001 - 0x7fff`
: Specification-defined key types. Key types defined by this standard always have bit 15 clear. Unallocated key type values in this range are reserved for future use.

`0x8000 - 0xffff`
: Implementation-defined key types. Implementations that define additional key types must use an encoding with bit 15 set. The related support macros will be easier to write if these key encodings also respect the bitwise structure used by standard encodings.

`PSA_KEY_TYPE_NONE` (macro)

An invalid key type value.

```
#define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x0000)
```

Zero is not the encoding of any key type.

9.2.2 Key categories

PSA_KEY_TYPE_IS_UNSTRUCTURED (macro)

Whether a key type is an unstructured array of bytes.

```
#define PSA_KEY_TYPE_IS_UNSTRUCTURED(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

Description

This encompasses both symmetric keys and non-key data.

See *Symmetric keys* on page 55 for a list of symmetric key types.

PSA_KEY_TYPE_IS_ASYMMETRIC (macro)

Whether a key type is asymmetric: either a key pair or a public key.

```
#define PSA_KEY_TYPE_IS_ASYMMETRIC(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

Description

See *RSA keys* on page 58 for a list of asymmetric key types.

PSA_KEY_TYPE_IS_PUBLIC_KEY (macro)

Whether a key type is the public part of a key pair.

```
#define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_IS_KEY_PAIR (macro)

Whether a key type is a key pair containing a private part and a public part.

```
#define PSA_KEY_TYPE_IS_KEY_PAIR(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

9.2.3 Symmetric keys

PSA_KEY_TYPE_RAW_DATA (macro)

Raw data.

```
#define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x1001)
```

A “key” of this type cannot be used for any cryptographic operation. Applications can use this type to store arbitrary data in the keystore.

The bit size of a raw key must be a non-zero multiple of 8. The maximum size of a raw key is IMPLEMENTATION DEFINED.

PSA_KEY_TYPE_HMAC (macro)

HMAC key.

```
#define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x1100)
```

The key policy determines which underlying hash algorithm the key can be used for.

The bit size of an HMAC key must be a non-zero multiple of 8. An HMAC key is typically the same size as the output of the underlying hash algorithm. An HMAC key that is longer than the block size of the underlying hash algorithm will be hashed before use.

When an HMAC key is created that is longer than the block size, it is IMPLEMENTATION DEFINED whether the implementation stores the original HMAC key, or the hash of the HMAC key. If the hash of the key is stored, the key size reported by `psa_get_key_attributes()` will be the size of the hashed key.

> **Note:**
>
> `PSA_HASH_LENGTH(alg)` provides the output size of hash algorithm `alg`, in bytes.
>
> `PSA_HASH_BLOCK_LENGTH(alg)` provides the block size of hash algorithm `alg`, in bytes.

PSA_KEY_TYPE_DERIVE (macro)

A secret for key derivation.

```
#define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x1200)
```

The key policy determines which key derivation algorithm the key can be used for.

The bit size of a secret for key derivation must be a non-zero multiple of 8. The maximum size of a secret for key derivation is IMPLEMENTATION DEFINED.

PSA_KEY_TYPE_AES (macro)

Key for a cipher, AEAD or MAC algorithm based on the AES block cipher.

```
#define PSA_KEY_TYPE_AES ((psa_key_type_t)0x2400)
```

The size of the key is related to the AES algorithm variant. For algorithms except the XTS block cipher mode, the following key sizes are used:

- AES-128 uses a 16-byte key : `key_bits = 128`
- AES-192 uses a 24-byte key : `key_bits = 192`
- AES-256 uses a 32-byte key : `key_bits = 256`

For the XTS block cipher mode (`PSA_ALG_XTS`), the following key sizes are used:

- AES-128-XTS uses two 16-byte keys : `key_bits = 256`
- AES-192-XTS uses two 24-byte keys : `key_bits = 384`
- AES-256-XTS uses two 32-byte keys : `key_bits = 512`

The AES block cipher is defined in *FIPS Publication 197: Advanced Encryption Standard (AES)* [FIPS197].

PSA_KEY_TYPE_DES (macro)

Key for a cipher or MAC algorithm based on DES or 3DES (Triple-DES).

```
#define PSA_KEY_TYPE_DES ((psa_key_type_t)0x2301)
```

The size of the key determines which DES algorithm is used:

- Single DES uses an 8-byte key : `key_bits = 64`
- 2-key 3DES uses a 16-byte key : `key_bits = 128`
- 3-key 3DES uses a 24-byte key : `key_bits = 192`

> **Warning:** Single DES and 2-key 3DES are weak and strongly deprecated and are only recommended for decrypting legacy data.
>
> 3-key 3DES is weak and deprecated and is only recommended for use in legacy protocols.

The DES and 3DES block ciphers are defined in *NIST Special Publication 800-67: Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher* [SP800-67].

PSA_KEY_TYPE_CAMELLIA (macro)

Key for a cipher, AEAD or MAC algorithm based on the Camellia block cipher.

```
#define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x2403)
```

The size of the key is related to the Camellia algorithm variant. For algorithms except the XTS block cipher mode, the following key sizes are used:

- Camellia-128 uses a 16-byte key : `key_bits = 128`
- Camellia-192 uses a 24-byte key : `key_bits = 192`
- Camellia-256 uses a 32-byte key : `key_bits = 256`

For the XTS block cipher mode (PSA_ALG_XTS), the following key sizes are used:

- Camellia-128-XTS uses two 16-byte keys : `key_bits = 256`
- Camellia-192-XTS uses two 24-byte keys : `key_bits = 384`
- Camellia-256-XTS uses two 32-byte keys : `key_bits = 512`

The Camellia block cipher is defined in *Specification of Camellia — a 128-bit Block Cipher* [NTT-CAM] and also described in *A Description of the Camellia Encryption Algorithm* [RFC3713].

PSA_KEY_TYPE_SM4 (macro)

Key for a cipher, AEAD or MAC algorithm based on the SM4 block cipher.

```
#define PSA_KEY_TYPE_SM4 ((psa_key_type_t)0x2405)
```

For algorithms except the XTS block cipher mode, the SM4 key size is 128 bits (16 bytes).

For the XTS block cipher mode (PSA_ALG_XTS), the SM4 key size is 256 bits (two 16-byte keys).

The SM4 block cipher is defined in *GB/T 32907-2016: Information security technology — SM4 block cipher algorithm* [PRC-SM4] and also described in *The SM4 Blockcipher Algorithm And Its Modes Of Operations* [IETF-SM4].

PSA_KEY_TYPE_ARC4 (macro)

Key for the ARC4 stream cipher.

```
#define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x2002)
```

> **Warning:** The ARC4 cipher is weak and deprecated and is only recommended for use in legacy protocols.

The ARC4 cipher supports key sizes between 40 and 2048 bits, that are multiples of 8. (5 to 256 bytes)

Use algorithm PSA_ALG_STREAM_CIPHER to use this key with the ARC4 cipher.

PSA_KEY_TYPE_CHACHA20 (macro)

Key for the ChaCha20 stream cipher or the ChaCha20-Poly1305 AEAD algorithm.

```
#define PSA_KEY_TYPE_CHACHA20 ((psa_key_type_t)0x2004)
```

The ChaCha20 key size is 256 bits (32 bytes).

- Use algorithm PSA_ALG_STREAM_CIPHER to use this key with the ChaCha20 cipher for unauthenticated encryption. See PSA_ALG_STREAM_CIPHER for details of this algorithm.
- Use algorithm PSA_ALG_CHACHA20_POLY1305 to use this key with the ChaCha20 cipher and Poly1305 authenticator for AEAD. See PSA_ALG_CHACHA20_POLY1305 for details of this algorithm.

9.2.4 RSA keys

PSA_KEY_TYPE_RSA_PUBLIC_KEY (macro)

RSA public key.

```
#define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x4001)
```

PSA_KEY_TYPE_RSA_KEY_PAIR (macro)

RSA key pair: both the private and public key.

```
#define PSA_KEY_TYPE_RSA_KEY_PAIR ((psa_key_type_t)0x7001)
```

PSA_KEY_TYPE_IS_RSA (macro)

Whether a key type is an RSA key. This includes both key pairs and public keys.

```
#define PSA_KEY_TYPE_IS_RSA(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

9.2.5 Elliptic Curve keys

`psa_ecc_family_t` (type)

The type of PSA elliptic curve family identifiers.

```
typedef uint8_t psa_ecc_family_t;
```

The curve identifier is required to create an ECC key using the `PSA_KEY_TYPE_ECC_KEY_PAIR()` or `PSA_KEY_TYPE_ECC_PUBLIC_KEY()` macros.

The specific ECC curve within a family is identified by the `key_bits` attribute of the key.

The range of Elliptic curve family identifier values is divided as follows:

`0x00 - 0x7f` ECC family identifiers defined by this standard. Unallocated values in this range are reserved for future use.

`0x80 - 0xff` Implementations that define additional families must use an encoding in this range.

PSA_KEY_TYPE_ECC_KEY_PAIR (macro)

Elliptic curve key pair: both the private and public key.

```
#define PSA_KEY_TYPE_ECC_KEY_PAIR(curve) /* specification-defined value */
```

Parameters

`curve`	A value of type `psa_ecc_family_t` that identifies the ECC curve family to be used.

PSA_KEY_TYPE_ECC_PUBLIC_KEY (macro)

Elliptic curve public key.

```
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) /* specification-defined value */
```

Parameters

`curve`	A value of type `psa_ecc_family_t` that identifies the ECC curve family to be used.

PSA_ECC_FAMILY_SECP_K1 (macro)

SEC Koblitz curves over prime fields.

```
#define PSA_ECC_FAMILY_SECP_K1 ((psa_ecc_family_t) 0x17)
```

This family comprises the following curves:

- secp192k1 : `key_bits = 192`
- secp224k1 : `key_bits = 225`
- secp256k1 : `key_bits = 256`

They are defined in *SEC 2: Recommended Elliptic Curve Domain Parameters* [SEC2].

PSA_ECC_FAMILY_SECP_R1 (macro)

SEC random curves over prime fields.

```
#define PSA_ECC_FAMILY_SECP_R1 ((psa_ecc_family_t) 0x12)
```

This family comprises the following curves:

- secp192r1 : `key_bits = 192`
- secp224r1 : `key_bits = 224`
- secp256r1 : `key_bits = 256`
- secp384r1 : `key_bits = 384`
- secp521r1 : `key_bits = 521`

They are defined in [SEC2].

PSA_ECC_FAMILY_SECP_R2 (macro)

> **Warning:** This family of curves is weak and deprecated.

```
#define PSA_ECC_FAMILY_SECP_R2 ((psa_ecc_family_t) 0x1b)
```

This family comprises the following curves:

- secp160r2 : `key_bits = 160` *(Deprecated)*

It is defined in the superseded *SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0* [SEC2v1].

PSA_ECC_FAMILY_SECT_K1 (macro)

SEC Koblitz curves over binary fields.

```
#define PSA_ECC_FAMILY_SECT_K1 ((psa_ecc_family_t) 0x27)
```

This family comprises the following curves:

- sect163k1 : `key_bits = 163` *(Deprecated)*
- sect233k1 : `key_bits = 233`
- sect239k1 : `key_bits = 239`
- sect283k1 : `key_bits = 283`
- sect409k1 : `key_bits = 409`
- sect571k1 : `key_bits = 571`

They are defined in [SEC2].

> **Warning:** The 163-bit curve sect163k1 is weak and deprecated and is only recommended for use in legacy protocols.

PSA_ECC_FAMILY_SECT_R1 (macro)

SEC random curves over binary fields.

```
#define PSA_ECC_FAMILY_SECT_R1 ((psa_ecc_family_t) 0x22)
```

This family comprises the following curves:

- sect163r1 : `key_bits = 163` *(Deprecated)*
- sect233r1 : `key_bits = 233`
- sect283r1 : `key_bits = 283`
- sect409r1 : `key_bits = 409`
- sect571r1 : `key_bits = 571`

They are defined in [SEC2].

> **Warning:** The 163-bit curve sect163r1 is weak and deprecated and is only recommended for use in legacy protocols.

PSA_ECC_FAMILY_SECT_R2 (macro)

SEC additional random curves over binary fields.

```
#define PSA_ECC_FAMILY_SECT_R2 ((psa_ecc_family_t) 0x2b)
```

This family comprises the following curves:

- sect163r2 : `key_bits = 163` *(Deprecated)*

It is defined in [SEC2].

> **Warning:** The 163-bit curve sect163r2 is weak and deprecated and is only recommended for use in legacy protocols.

PSA_ECC_FAMILY_BRAINPOOL_P_R1 (macro)

Brainpool P random curves.

```
#define PSA_ECC_FAMILY_BRAINPOOL_P_R1 ((psa_ecc_family_t) 0x30)
```

This family comprises the following curves:

- brainpoolP160r1 : `key_bits = 160` *(Deprecated)*
- brainpoolP192r1 : `key_bits = 192`
- brainpoolP224r1 : `key_bits = 224`
- brainpoolP256r1 : `key_bits = 256`
- brainpoolP320r1 : `key_bits = 320`
- brainpoolP384r1 : `key_bits = 384`
- brainpoolP512r1 : `key_bits = 512`

They are defined in *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation* [RFC5639].

> **Warning:** The 160-bit curve brainpoolP160r1 is weak and deprecated and is only recommended for use in legacy protocols.

PSA_ECC_FAMILY_FRP (macro)

Curve used primarily in France and elsewhere in Europe.

```
#define PSA_ECC_FAMILY_FRP ((psa_ecc_family_t) 0x33)
```

This family comprises one 256-bit curve:

- FRP256v1 : `key_bits = 256`

This is defined by *Publication d'un paramétrage de courbe elliptique visant des applications de passeport électronique et de l'administration électronique française* [FRP].

PSA_ECC_FAMILY_MONTGOMERY (macro)

Montgomery curves.

```
#define PSA_ECC_FAMILY_MONTGOMERY ((psa_ecc_family_t) 0x41)
```

This family comprises the following Montgomery curves:

- Curve25519 : `key_bits = 255`
- Curve448 : `key_bits = 448`

Keys in this family can only be used with the `PSA_ALG_ECDH` key agreement algorithm.

Curve25519 is defined in *Curve25519: new Diffie-Hellman speed records* [Curve25519]. Curve448 is defined in *Ed448-Goldilocks, a new elliptic curve* [Curve448].

PSA_KEY_TYPE_IS_ECC (macro)

Whether a key type is an elliptic curve key, either a key pair or a public key.

```
#define PSA_KEY_TYPE_IS_ECC(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_IS_ECC_KEY_PAIR (macro)

Whether a key type is an elliptic curve key pair.

```
#define PSA_KEY_TYPE_IS_ECC_KEY_PAIR(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY (macro)

Whether a key type is an elliptic curve public key.

```
#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_ECC_GET_FAMILY (macro)

Extract the curve family from an elliptic curve key type.

```
#define PSA_KEY_TYPE_ECC_GET_FAMILY(type) /* specification-defined value */
```

Parameters

`type`	An elliptic curve key type (value of type `psa_key_type_t` such that `PSA_KEY_TYPE_IS_ECC(type)` is true).

Returns: `psa_ecc_family_t`

The elliptic curve family id, if `type` is a supported elliptic curve key. Unspecified if `type` is not a supported elliptic curve key.

9.2.6 Diffie Hellman keys

`psa_dh_family_t` (type)

The type of PSA finite-field Diffie-Hellman group family identifiers.

```
typedef uint8_t psa_dh_family_t;
```

The group family identifier is required to create a finite-field Diffie-Hellman key using the `PSA_KEY_TYPE_DH_KEY_PAIR()` or `PSA_KEY_TYPE_DH_PUBLIC_KEY()` macros.

The specific Diffie-Hellman group within a family is identified by the `key_bits` attribute of the key.

The range of Diffie-Hellman group family identifier values is divided as follows:

`0x00 – 0x7f` DH group family identifiers defined by this standard. Unallocated values in this range are reserved for future use.

`0x80 – 0xff` Implementations that define additional families must use an encoding in this range.

PSA_KEY_TYPE_DH_KEY_PAIR (macro)

Finite-field Diffie-Hellman key pair: both the private key and public key.

```
#define PSA_KEY_TYPE_DH_KEY_PAIR(group) /* specification-defined value */
```

Parameters

`group`	A value of type `psa_dh_family_t` that identifies the Diffie-Hellman group family to be used.

PSA_KEY_TYPE_DH_PUBLIC_KEY (macro)

Finite-field Diffie-Hellman public key.

```
#define PSA_KEY_TYPE_DH_PUBLIC_KEY(group) /* specification-defined value */
```

Parameters

`group`	A value of type `psa_dh_family_t` that identifies the Diffie-Hellman group family to be used.

PSA_DH_FAMILY_RFC7919 (macro)

Finite-field Diffie-Hellman groups defined for TLS in RFC 7919.

```
#define PSA_DH_FAMILY_RFC7919 ((psa_dh_family_t) 0x03)
```

This family includes groups with the following key sizes (in bits): 2048, 3072, 4096, 6144, 8192. An implementation can support all of these sizes or only a subset.

Keys is this group can only be used with the PSA_ALG_FFDH key agreement algorithm.

These groups are defined by *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)* [RFC7919] Appendix A.

PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY (macro)

The key pair type corresponding to a public key type.

```
#define PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type) \
    /* specification-defined value */
```

Parameters

type	A public key type or key pair type.

Returns

The corresponding key pair type. If type is not a public key or a key pair, the return value is undefined.

Description

If type is a key pair type, it will be left unchanged.

PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR (macro)

The public key type corresponding to a key pair type.

```
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) \
    /* specification-defined value */
```

Parameters

type	A public key type or key pair type.

Returns

The corresponding public key type. If type is not a public key or a key pair, the return value is undefined.

Description

If type is a public key type, it will be left unchanged.

PSA_KEY_TYPE_IS_DH (macro)

Whether a key type is a Diffie-Hellman key, either a key pair or a public key.

```
#define PSA_KEY_TYPE_IS_DH(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_IS_DH_KEY_PAIR (macro)

Whether a key type is a Diffie-Hellman key pair.

```
#define PSA_KEY_TYPE_IS_DH_KEY_PAIR(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_IS_DH_PUBLIC_KEY (macro)

Whether a key type is a Diffie-Hellman public key.

```
#define PSA_KEY_TYPE_IS_DH_PUBLIC_KEY(type) /* specification-defined value */
```

Parameters

`type`	A key type (value of type `psa_key_type_t`).

PSA_KEY_TYPE_DH_GET_FAMILY (macro)

Extract the group family from a Diffie-Hellman key type.

```
#define PSA_KEY_TYPE_DH_GET_FAMILY(type) /* specification-defined value */
```

Parameters

`type`	A Diffie-Hellman key type (value of type `psa_key_type_t` such that `PSA_KEY_TYPE_IS_DH(type)` is true).

Returns: `psa_dh_family_t`

The Diffie-Hellman group family id, if `type` is a supported Diffie-Hellman key. Unspecified if `type` is not a supported Diffie-Hellman key.

9.2.7 Attribute accessors

psa_set_key_type (function)

Declare the type of a key.

```
void psa_set_key_type(psa_key_attributes_t * attributes,
                      psa_key_type_t type);
```

Parameters

`attributes`	The attribute object to write to.
`type`	The key type to write. If this is `PSA_KEY_TYPE_NONE`, the key type in `attributes` becomes unspecified.

Returns: `void`

Description

This function overwrites any key type previously set in `attributes`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:
>
> - This function can be declared as `static` or `inline`, instead of using the default external linkage.
> - This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

`psa_get_key_type` (function)

Retrieve the key type from key attributes.

```
psa_key_type_t psa_get_key_type(const psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The key attribute object to query.

Returns: `psa_key_type_t`

The key type stored in the attribute object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:
>
> - This function can be declared as `static` or `inline`, instead of using the default external linkage.
> - This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

`psa_get_key_bits` (function)

Retrieve the key size from key attributes.

```
size_t psa_get_key_bits(const psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The key attribute object to query.

Returns: `size_t`

The key size stored in the attribute object, in bits.

Description

Implementation note

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as `static` or `inline`, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

`psa_set_key_bits` (function)

Declare the size of a key.

```
void psa_set_key_bits(psa_key_attributes_t * attributes,
                      size_t bits);
```

Parameters

`attributes`	The attribute object to write to.
`bits`	The key size in bits. If this is `0`, the key size in `attributes` becomes unspecified. Keys of size `0` are not supported.

Returns: `void`

Description

This function overwrites any key size previously set in `attributes`.

Implementation note

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as `static` or `inline`, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

9.3 Key lifetimes

The lifetime of a key indicates where it is stored and which application and system actions will create and destroy it.

Lifetime values are composed from:

- A persistence level, which indicates what device management actions can cause it to be destroyed. In particular, it indicates whether the key is volatile or persistent. See `psa_key_persistence_t` for more information.
- A location indicator, which indicates where the key is stored and where operations on the key are performed. See `psa_key_location_t` for more information.

There are two main types of lifetime, indicated by the persistence level: *volatile* and *persistent*.

9.3.1 Volatile keys

Volatile keys are automatically destroyed when the application instance terminates or on a power reset of the device. Volatile keys can be explicitly destroyed by the application.

Conceptually, a volatile key is stored in RAM. Volatile keys have the lifetime `PSA_KEY_LIFETIME_VOLATILE`.

To create a volatile key:

1. Populate a `psa_key_attributes_t` object with the required type, size, policy and other key attributes.
2. Create the key with one of the key creation functions. If successful, these functions output a transient key identifier.

To destroy a volatile key: call `psa_destroy_key()` with the key identifier. There must be a matching call to `psa_destroy_key()` for each successful call to a create a volatile key.

9.3.2 Persistent keys

Persistent keys are preserved until the application explicitly destroys them or until an implementation-specific device management event occurs, for example, a factory reset.

Each persistent key has a permanent key identifier, which acts as a name for the key. Within an application, the key identifier corresponds to a single key. The application specifies the key identifier when the key is created and when using the key.

The lifetime attribute of a persistent key indicates how and where it is stored. The default lifetime value for a persistent key is `PSA_KEY_LIFETIME_PERSISTENT`, which corresponds to a default storage area. This specification defines how implementations can provide other 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. Populate a `psa_key_attributes_t` object with the key's type, size, policy and other attributes.
2. In the attributes object, set the desired lifetime and persistent identifier for the key.
3. Create the key with one of the key creation functions. If successful, these functions output the key identifier that was specified by the application in step 2.

To access an existing persistent key: use the key identifier in any API that requires a key.

To destroy a persistent key: call psa_destroy_key() with the key identifier. Destroying a persistent key permanently removes it from memory and storage.

By default, persistent key material is removed from volatile memory when not in use. Frequently used persistent keys can benefit from caching, depending on the implementation and the application. Caching can be enabled by creating the key with the PSA_KEY_USAGE_CACHE policy. Cached keys can be removed from volatile memory by calling psa_purge_key(). See also *Memory cleanup* on page 38 and *Managing key material* on page 39.

9.3.3 Lifetime encodings

`psa_key_lifetime_t` (type)

Encoding of key lifetimes.

```
typedef uint32_t psa_key_lifetime_t;
```

The lifetime of a key indicates where it is stored and which application and system actions will create and destroy it.

Lifetime values have the following structure:

Bits[7:0]: Persistence level

This value indicates what device management actions can cause it to be destroyed. In particular, it indicates whether the key is *volatile* or *persistent*. See psa_key_persistence_t for more information.

PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) returns the persistence level for a key lifetime value.

Bits[31:8]: Location indicator

This value indicates where the key material is stored (or at least where it is accessible in cleartext) and where operations on the key are performed. See psa_key_location_t for more information.

PSA_KEY_LIFETIME_GET_LOCATION(lifetime) returns the location indicator for a key lifetime value.

Volatile keys are automatically destroyed when the application instance terminates or on a power reset of the device. Persistent keys are preserved until the application explicitly destroys them or until an implementation-specific device management event occurs, for example, a factory reset.

Persistent keys have a key identifier of type psa_key_id_t. This identifier remains valid throughout the lifetime of the key, even if the application instance that created the key terminates.

This specification defines two basic lifetime values:

- Keys with the lifetime PSA_KEY_LIFETIME_VOLATILE are volatile. All implementations should support this lifetime.
- Keys with the lifetime PSA_KEY_LIFETIME_PERSISTENT are persistent. All implementations that have access to persistent storage with appropriate security guarantees should support this lifetime.

`psa_key_persistence_t` (type)

Encoding of key persistence levels.

```
typedef uint8_t psa_key_persistence_t;
```

What distinguishes different persistence levels is which device management events can cause keys to be destroyed. For example, power reset, transfer of device ownership, or a factory reset are device management events that can affect keys at different persistence levels. The specific management events which affect persistent keys at different levels is outside the scope of the PSA Cryptography specification.

Values for persistence levels defined by this specification are shown in Table 4.

Table 4 Key persistence level values

Persistence level	Definition
`0 = PSA_KEY_PERSISTENCE_VOLATILE`	Volatile key. A volatile key is automatically destroyed by the implementation when the application instance terminates. In particular, a volatile key is automatically destroyed on a power reset of the device.
`1 = PSA_KEY_PERSISTENCE_DEFAULT`	Persistent key with a default lifetime. Implementations should support this value if they support persistent keys at all. Applications should use this value if they have no specific needs that are only met by implementation-specific features.
`2 - 127`	Persistent key with a PSA-specified lifetime. The PSA Cryptography specification does not define the meaning of these values, but other PSA specifications may do so.
`128 - 254`	Persistent key with a vendor-specified lifetime. No PSA specification will define the meaning of these values, so implementations may choose the meaning freely. As a guideline, higher persistence levels should cause a key to survive more management events than lower levels.
`255 = PSA_KEY_PERSISTENCE_READ_ONLY`	Read-only or write-once key. A key with this persistence level cannot be destroyed. Implementations that support such keys may either allow their creation through the PSA Cryptography API, preferably only to applications with the appropriate privilege, or only expose keys created through implementation-specific means such as a factory ROM engraving process. Note that keys that are read-only due to policy restrictions rather than due to physical limitations should not have this persistence level.

Note:

Key persistence levels are 8-bit values. Key management interfaces operate on lifetimes (type psa_key_lifetime_t), and encode the persistence value as the lower 8 bits of a 32-bit value.

psa_key_location_t (type)

Encoding of key location indicators.

```
typedef uint32_t psa_key_location_t;
```

If an implementation of this API can make calls to external cryptoprocessors such as secure elements, the location of a key indicates which secure element performs the operations on the key. If the key material is not stored persistently inside the secure element, it must be stored in a wrapped form such that only the secure element can access the key material in cleartext.

Values for location indicators defined by this specification are shown in Table 5.

Table 5 Key location indicator values

Location indicator	Definition
0	Primary local storage. All implementations should support this value. The primary local storage is typically the same storage area that contains the key metadata.
1	Primary secure element. Implementations should support this value if there is a secure element attached to the operating environment. As a guideline, secure elements may provide higher resistance against side channel and physical attacks than the primary local storage, but may have restrictions on supported key types, sizes, policies and operations and may have different performance characteristics.
2 - 0x7fffff	Other locations defined by a PSA specification. The PSA Cryptography API does not currently assign any meaning to these locations, but future versions of this specification or other PSA specifications may do so.
0x800000 - 0xffffff	Vendor-defined locations. No PSA specification will assign a meaning to locations in this range.

Note:

Key location indicators are 24-bit values. Key management interfaces operate on lifetimes (type psa_key_lifetime_t), and encode the location as the upper 24 bits of a 32-bit value.

9.3.4 Lifetime values

PSA_KEY_LIFETIME_VOLATILE (macro)

The default lifetime for volatile keys.

```
#define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t) 0x00000000)
```

A volatile key only exists as long as its identifier is not destroyed. The key material is guaranteed to be erased on a power reset.

A key with this lifetime is typically stored in the RAM area of the PSA Crypto subsystem. However this is an implementation choice. If an implementation stores data about the key in a non-volatile memory, it must release all the resources associated with the key and erase the key material if the calling application terminates.

PSA_KEY_LIFETIME_PERSISTENT (macro)

The default lifetime for persistent keys.

```
#define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t) 0x00000001)
```

A persistent key remains in storage until it is explicitly destroyed or until the corresponding storage area is wiped. This specification does not define any mechanism to wipe a storage area. Implementations are permitted to provide their own mechanism, for example, to perform a factory reset, to prepare for device refurbishment, or to uninstall an application.

This lifetime value is the default storage area for the calling application. Implementations can offer other storage areas designated by other lifetime values as implementation-specific extensions.

PSA_KEY_PERSISTENCE_VOLATILE (macro)

The persistence level of volatile keys.

```
#define PSA_KEY_PERSISTENCE_VOLATILE ((psa_key_persistence_t) 0x00)
```

See psa_key_persistence_t for more information.

PSA_KEY_PERSISTENCE_DEFAULT (macro)

The default persistence level for persistent keys.

```
#define PSA_KEY_PERSISTENCE_DEFAULT ((psa_key_persistence_t) 0x01)
```

See psa_key_persistence_t for more information.

PSA_KEY_PERSISTENCE_READ_ONLY (macro)

A persistence level indicating that a key is never destroyed.

```
#define PSA_KEY_PERSISTENCE_READ_ONLY ((psa_key_persistence_t) 0xff)
```

See psa_key_persistence_t for more information.

PSA_KEY_LOCATION_LOCAL_STORAGE (macro)

The local storage area for persistent keys.

```
#define PSA_KEY_LOCATION_LOCAL_STORAGE ((psa_key_location_t) 0x000000)
```

This storage area is available on all systems that can store persistent keys without delegating the storage to a third-party cryptoprocessor.

See psa_key_location_t for more information.

PSA_KEY_LOCATION_PRIMARY_SECURE_ELEMENT (macro)

The default secure element storage area for persistent keys.

```
#define PSA_KEY_LOCATION_PRIMARY_SECURE_ELEMENT ((psa_key_location_t) 0x000001)
```

This storage location is available on systems that have one or more secure elements that are able to store keys.

Vendor-defined locations must be provided by the system for storing keys in additional secure elements.

See psa_key_location_t for more information.

9.3.5 Attribute accessors

psa_set_key_lifetime (function)

Set the location of a persistent key.

```
void psa_set_key_lifetime(psa_key_attributes_t * attributes,
                          psa_key_lifetime_t lifetime);
```

Parameters

attributes	The attribute object to write to.
lifetime	The lifetime for the key. If this is PSA_KEY_LIFETIME_VOLATILE, the key will be volatile, and the key identifier attribute is reset to PSA_KEY_ID_NULL.

Returns: void

Description

To make a key persistent, give it a persistent key identifier by using psa_set_key_id(). By default, a key that has a persistent identifier is stored in the default storage area identifier by PSA_KEY_LIFETIME_PERSISTENT. Call this function to choose a storage area, or to explicitly declare the key as volatile.

This function does not access storage, it merely stores the given value in the attribute object. The persistent key will be written to storage when the attribute object is passed to a key creation function such as psa_import_key(), psa_generate_key(), psa_key_derivation_output_key() or psa_copy_key().

Implementation note

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as `static` or `inline`, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

psa_get_key_lifetime (function)

Retrieve the lifetime from key attributes.

```
psa_key_lifetime_t psa_get_key_lifetime(const psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The key attribute object to query.

Returns: `psa_key_lifetime_t`

The lifetime value stored in the attribute object.

Description

Implementation note

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as `static` or `inline`, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

9.3.6 Support macros

PSA_KEY_LIFETIME_GET_PERSISTENCE (macro)

Extract the persistence level from a key lifetime.

```
#define PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) \
    ((psa_key_persistence_t) ((lifetime) & 0x000000ff))
```

Parameters

`lifetime`	The lifetime value to query (value of type `psa_key_lifetime_t`).

PSA_KEY_LIFETIME_GET_LOCATION (macro)

Extract the location indicator from a key lifetime.

```
#define PSA_KEY_LIFETIME_GET_LOCATION(lifetime) \
    ((psa_key_location_t) ((lifetime) >> 8))
```

Parameters

`lifetime`	The lifetime value to query (value of type `psa_key_lifetime_t`).

PSA_KEY_LIFETIME_IS_VOLATILE (macro)

Whether a key lifetime indicates that the key is volatile.

```
#define PSA_KEY_LIFETIME_IS_VOLATILE(lifetime) \
    (PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) == PSA_KEY_PERSISTENCE_VOLATILE)
```

Parameters

`lifetime`	The lifetime value to query (value of type `psa_key_lifetime_t`).

Returns

`1` if the key is volatile, otherwise `0`.

Description

A volatile key is automatically destroyed by the implementation when the application instance terminates. In particular, a volatile key is automatically destroyed on a power reset of the device.

A key that is not volatile is persistent. Persistent keys are preserved until the application explicitly destroys them or until an implementation-specific device management event occurs, for example, a factory reset.

PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION (macro)

Construct a lifetime from a persistence level and a location.

```
#define PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(persistence, location) \
    ((location) << 8 | (persistence))
```

Parameters

`persistence`	The persistence level (value of type `psa_key_persistence_t`).
`location`	The location indicator (value of type `psa_key_location_t`).

Returns

The constructed lifetime value.

9.4 Key identifiers

Key identifiers are integral values that act as permanent names for persistent keys, or as transient references to volatile keys. Key identifiers use the `psa_key_id_t` type, and the range of identifier values is divided as follows:

`PSA_KEY_ID_NULL = 0`
: Reserved as an invalid key identifier.

`PSA_KEY_ID_USER_MIN - PSA_KEY_ID_USER_MAX`
: Applications can freely choose persistent key identifiers in this range.

`PSA_KEY_ID_VENDOR_MIN - PSA_KEY_ID_VENDOR_MAX`
: Implementations can define additional persistent key identifiers in this range, and must allocate any volatile key identifiers from this range.

Key identifiers outside these ranges are reserved for future use.

Key identifiers are output from a successful call to one of the key creation functions. For persistent keys, this is the same identifier as the one specified in the key attributes used to create the key. The key identifier remains valid until it is invalidated by passing it to `psa_destroy_key()`. A volatile key identifier must not be used after it has been invalidated.

If an invalid key identifier is provided as a parameter in any function, the function will return `PSA_ERROR_INVALID_HANDLE`; except for the special case of calling `psa_destroy_key(PSA_KEY_ID_NULL)`, which has no effect and always returns `PSA_SUCCESS`.

Valid key identifiers must have distinct values within the same application. If the implementation provides caller isolation, then key identifiers are local to each application. That is, the same key identifier in two applications corresponds to two different keys.

9.4.1 Key identifier type

psa_key_id_t (type)

Key identifier.

```
typedef uint32_t psa_key_id_t;
```

A key identifier can be a permanent name for a persistent key, or a transient reference to volatile key. See *Key identifiers* on page 75.

PSA_KEY_ID_NULL (macro)

The null key identifier.

```
#define PSA_KEY_ID_NULL ((psa_key_id_t)0)
```

The null key identifier is always invalid, except when used without in a call to `psa_destroy_key()` which will return `PSA_SUCCESS`.

PSA_KEY_ID_USER_MIN (macro)

The minimum value for a key identifier chosen by the application.

```
#define PSA_KEY_ID_USER_MIN ((psa_key_id_t)0x00000001)
```

PSA_KEY_ID_USER_MAX (macro)

The maximum value for a key identifier chosen by the application.

```
#define PSA_KEY_ID_USER_MAX ((psa_key_id_t)0x3fffffff)
```

PSA_KEY_ID_VENDOR_MIN (macro)

The minimum value for a key identifier chosen by the implementation.

```
#define PSA_KEY_ID_VENDOR_MIN ((psa_key_id_t)0x40000000)
```

PSA_KEY_ID_VENDOR_MAX (macro)

The maximum value for a key identifier chosen by the implementation.

```
#define PSA_KEY_ID_VENDOR_MAX ((psa_key_id_t)0x7fffffff)
```

9.4.2 Attribute accessors

psa_set_key_id (function)

Declare a key as persistent and set its key identifier.

```
void psa_set_key_id(psa_key_attributes_t * attributes,
                    psa_key_id_t id);
```

Parameters

`attributes`	The attribute object to write to.
`id`	The persistent identifier for the key.

Returns: `void`

Description

The application must choose a value for `id` between `PSA_KEY_ID_USER_MIN` and `PSA_KEY_ID_USER_MAX`.

If the attribute object currently declares the key as volatile, which is the default lifetime of an attribute object, this function sets the lifetime attribute to `PSA_KEY_LIFETIME_PERSISTENT`.

This function does not access storage, it merely stores the given value in the attribute object. The persistent key will be written to storage when the attribute object is passed to a key creation function such as `psa_import_key()`, `psa_generate_key()`, `psa_key_derivation_output_key()` or `psa_copy_key()`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:
>
> - This function can be declared as `static` or `inline`, instead of using the default external linkage.
> - This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

psa_get_key_id (function)

Retrieve the key identifier from key attributes.

```
psa_key_id_t psa_get_key_id(const psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The key attribute object to query.

Returns: `psa_key_id_t`

The persistent identifier stored in the attribute object. This value is unspecified if the attribute object declares the key as volatile.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:
>
> - This function can be declared as `static` or `inline`, instead of using the default external linkage.
> - This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

9.5 Key policies

All keys have an associated policy that regulates which operations are permitted on the key. A key policy is composed of two elements:

- A set of usage flags. See *Key usage flags* on page 80.
- A specific algorithm that is permitted with the key. See *Permitted algorithms*.

The policy is part of the key attributes that are managed by a `psa_key_attributes_t` object.

A highly constrained implementation might not be able to support all the policies that can be expressed through this interface. If an implementation cannot create a key with the required policy, it must return an appropriate error code when the key is created.

9.5.1 Permitted algorithms

The permitted algorithm is encoded using a algorithm identifier, as described in *Algorithms* on page 98.

This specification only defines policies that restrict keys to a single algorithm, which is consistent with both common practice and security good practice.

The following algorithm policies are supported:

- `PSA_ALG_NONE` does not allow any cryptographic operation with the key. The key can still be used for non-cryptographic actions such as exporting, if permitted by the usage flags.
- A specific algorithm value permits this particular algorithm.
- A signature algorithm wildcard built from `PSA_ALG_ANY_HASH` allows the specified signature scheme with any hash algorithm.

When a key is used in a cryptographic operation, the application must supply the algorithm to use for the operation. This algorithm is checked against the key's permitted algorithm policy.

`psa_set_key_algorithm` (function)

Declare the permitted algorithm policy for a key.

```
void psa_set_key_algorithm(psa_key_attributes_t * attributes,
                           psa_algorithm_t alg);
```

Parameters

`attributes`	The attribute object to write to.
`alg`	The permitted algorithm to write.

Returns: `void`

Description

The permitted algorithm policy of a key encodes which algorithm or algorithms are permitted to be used with this key.

This function overwrites any permitted algorithm policy previously set in `attributes`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:
>
> - This function can be declared as `static` or `inline`, instead of using the default external linkage.
> - This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

`psa_get_key_algorithm` (function)

Retrieve the permitted algorithm policy from key attributes.

```
psa_algorithm_t psa_get_key_algorithm(const psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The key attribute object to query.

Returns: `psa_algorithm_t`

The algorithm stored in the attribute object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:
>
> - This function can be declared as `static` or `inline`, instead of using the default external linkage.
> - This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

9.5.2 Key usage flags

The usage flags are encoded in a bitmask, which has the type psa_key_usage_t. Four kinds of usage flag can be specified:

- The extractable flag PSA_KEY_USAGE_EXPORT determines whether the key material can be extracted.
- The copyable flag PSA_KEY_USAGE_COPY determines whether the key material can be copied into a new key, which can have a different lifetime or a more restrictive policy.
- The cacheable flag PSA_KEY_USAGE_CACHE determines whether the implementation is permitted to retain non-essential copies of the key material in RAM. This policy only applies to persistent keys. See also *Managing key material* on page 39.
- The other usage flags, for example, PSA_KEY_USAGE_ENCRYPT and PSA_KEY_USAGE_SIGN_MESSAGE, determine whether the corresponding operation is permitted on the key.

psa_key_usage_t (type)

Encoding of permitted usage on a key.

```
typedef uint32_t psa_key_usage_t;
```

PSA_KEY_USAGE_EXPORT (macro)

Permission to export the key.

```
#define PSA_KEY_USAGE_EXPORT ((psa_key_usage_t)0x00000001)
```

This flag allows the use of psa_export_key() to export a key from the cryptoprocessor. A public key or the public part of a key pair can always be exported regardless of the value of this permission flag.

This flag can also be required to copy a key using psa_copy_key() outside of a secure element. See also PSA_KEY_USAGE_COPY.

If a key does not have export permission, implementations must not allow the key to be exported in plain form from the cryptoprocessor, whether through psa_export_key() or through a proprietary interface. The key might still be exportable in a wrapped form, i.e. in a form where it is encrypted by another key.

PSA_KEY_USAGE_COPY (macro)

Permission to copy the key.

```
#define PSA_KEY_USAGE_COPY ((psa_key_usage_t)0x00000002)
```

This flag allows the use of psa_copy_key() to make a copy of the key with the same policy or a more restrictive policy.

For lifetimes for which the key is located in a secure element which enforce the non-exportability of keys, copying a key outside the secure element also requires the usage flag PSA_KEY_USAGE_EXPORT. Copying the key inside the secure element is permitted with just PSA_KEY_USAGE_COPY if the secure element supports it. For keys with the lifetime PSA_KEY_LIFETIME_VOLATILE or PSA_KEY_LIFETIME_PERSISTENT, the usage flag PSA_KEY_USAGE_COPY is sufficient to permit the copy.

PSA_KEY_USAGE_CACHE (macro)

Permission for the implementation to cache the key.

```
#define PSA_KEY_USAGE_CACHE ((psa_key_usage_t)0x00000004)
```

This flag allows the implementation to make additional copies of the key material that are not in storage and not for the purpose of an ongoing operation. Applications can use it as a hint to keep the key around for repeated access.

An application can request that cached key material is removed from memory by calling `psa_purge_key()`.

The presence of this usage flag when creating a key is a hint:

- An implementation is not required to cache keys that have this usage flag.
- An implementation must not report an error if it does not cache keys.

If this usage flag is not present, the implementation must ensure key material is removed from memory as soon as it is not required for an operation or for maintenance of a volatile key.

This flag must be preserved when reading back the attributes for all keys, regardless of key type or implementation behavior.

See also *Managing key material* on page 39.

PSA_KEY_USAGE_ENCRYPT (macro)

Permission to encrypt a message with the key.

```
#define PSA_KEY_USAGE_ENCRYPT ((psa_key_usage_t)0x00000100)
```

This flag allows the key to be used for a symmetric encryption operation, for an AEAD encryption-and-authentication operation, or for an asymmetric encryption operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- `psa_cipher_encrypt()`
- `psa_cipher_encrypt_setup()`
- `psa_aead_encrypt()`
- `psa_aead_encrypt_setup()`
- `psa_asymmetric_encrypt()`

For a key pair, this concerns the public key.

PSA_KEY_USAGE_DECRYPT (macro)

Permission to decrypt a message with the key.

```
#define PSA_KEY_USAGE_DECRYPT ((psa_key_usage_t)0x00000200)
```

This flag allows the key to be used for a symmetric decryption operation, for an AEAD decryption-and-verification operation, or for an asymmetric decryption operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- psa_cipher_decrypt()
- psa_cipher_decrypt_setup()
- psa_aead_decrypt()
- psa_aead_decrypt_setup()
- psa_asymmetric_decrypt()

For a key pair, this concerns the private key.

PSA_KEY_USAGE_SIGN_MESSAGE (macro)

Permission to sign a message with the key.

```
#define PSA_KEY_USAGE_SIGN_MESSAGE ((psa_key_usage_t)0x00000400)
```

This flag allows the key to be used for a MAC calculation operation or for an asymmetric message signature operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- psa_mac_compute()
- psa_mac_sign_setup()
- psa_sign_message()

For a key pair, this concerns the private key.

PSA_KEY_USAGE_VERIFY_MESSAGE (macro)

Permission to verify a message signature with the key.

```
#define PSA_KEY_USAGE_VERIFY_MESSAGE ((psa_key_usage_t)0x00000800)
```

This flag allows the key to be used for a MAC verification operation or for an asymmetric message signature verification operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used with the following APIs:

- psa_mac_verify()
- psa_mac_verify_setup()
- psa_verify_message()

For a key pair, this concerns the public key.

PSA_KEY_USAGE_SIGN_HASH (macro)

Permission to sign a message hash with the key.

```
#define PSA_KEY_USAGE_SIGN_HASH ((psa_key_usage_t)0x00001000)
```

This flag allows the key to be used to sign a message hash as part of an asymmetric signature operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used when calling `psa_sign_hash()`.

This flag automatically sets `PSA_KEY_USAGE_SIGN_MESSAGE`: if an application sets the flag `PSA_KEY_USAGE_SIGN_HASH` when creating a key, then the key always has the permissions conveyed by `PSA_KEY_USAGE_SIGN_MESSAGE`, and the flag `PSA_KEY_USAGE_SIGN_MESSAGE` will also be present when the application queries the usage flags of the key.

For a key pair, this concerns the private key.

PSA_KEY_USAGE_VERIFY_HASH (macro)

Permission to verify a message hash with the key.

```
#define PSA_KEY_USAGE_VERIFY_HASH ((psa_key_usage_t)0x00002000)
```

This flag allows the key to be used to verify a message hash as part of an asymmetric signature verification operation, if otherwise permitted by the key's type and policy. The flag must be present on keys used when calling `psa_verify_hash()`.

This flag automatically sets `PSA_KEY_USAGE_VERIFY_MESSAGE`: if an application sets the flag `PSA_KEY_USAGE_VERIFY_HASH` when creating a key, then the key always has the permissions conveyed by `PSA_KEY_USAGE_VERIFY_MESSAGE`, and the flag `PSA_KEY_USAGE_VERIFY_MESSAGE` will also be present when the application queries the usage flags of the key.

For a key pair, this concerns the public key.

PSA_KEY_USAGE_DERIVE (macro)

Permission to derive other keys from this key.

```
#define PSA_KEY_USAGE_DERIVE ((psa_key_usage_t)0x00004000)
```

This flag allows the key to be used for a key derivation operation or for a key agreement operation, if otherwise permitted by by the key's type and policy. The flag must be present on keys used with the following APIs:

- `psa_key_derivation_input_key()`
- `psa_key_derivation_key_agreement()`
- `psa_raw_key_agreement()`

psa_set_key_usage_flags (function)

Declare usage flags for a key.

```
void psa_set_key_usage_flags(psa_key_attributes_t * attributes,
                             psa_key_usage_t usage_flags);
```

Parameters

`attributes`	The attribute object to write to.
`usage_flags`	The usage flags to write.

Returns: `void`

Description

Usage flags are part of a key's policy. They encode what kind of operations are permitted on the key. For more details, see *Key policies* on page 78.

This function overwrites any usage flags previously set in `attributes`.

Implementation note

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as `static` or `inline`, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

`psa_get_key_usage_flags` (function)

Retrieve the usage flags from key attributes.

```
psa_key_usage_t psa_get_key_usage_flags(const psa_key_attributes_t * attributes);
```

Parameters

`attributes`	The key attribute object to query.

Returns: `psa_key_usage_t`

The usage flags stored in the attribute object.

Description

Implementation note

This is a simple accessor function that is not required to validate its inputs. The following approaches can be used to provide an efficient implementation:

- This function can be declared as `static` or `inline`, instead of using the default external linkage.
- This function can be provided as a function-like macro. In this form, the macro must evaluate each of its arguments exactly once, as if it was a function call.

9.6 Key management functions

9.6.1 Key creation

New keys can be created in the following ways:

- `psa_import_key()` creates a key from a data buffer provided by the application.
- `psa_generate_key()` creates a key from randomly generated data.

- `psa_key_derivation_output_key()` creates a key from data generated by a pseudorandom derivation process. See *Key derivation* on page 181.
- `psa_copy_key()` duplicates an existing key with a different lifetime or with a more restrictive usage policy.

When creating a key, the attributes for the new key are specified in a `psa_key_attributes_t` object. Each key creation function defines how it uses the attributes.

> **Note:**
>
> The attributes for a key are immutable after the key has been created.
>
> The application must set the key algorithm policy and the appropriate key usage flags in the attributes in order for the key to be used in any cryptographic operations.

`psa_import_key` (function)

Import a key in binary format.

```
psa_status_t psa_import_key(const psa_key_attributes_t * attributes,
                            const uint8_t * data,
                            size_t data_length,
                            psa_key_id_t * key);
```

Parameters

`attributes`

The attributes for the new key. This function uses the attributes as follows:

- The key type is required, and determines how the `data` buffer is interpreted.
- The key size is always determined from the `data` buffer. If the key size in `attributes` is nonzero, it must be equal to the size determined from `data`.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see *Permitted algorithms* on page 78.
- The key usage flags define what operations are permitted with the key, see *Key usage flags* on page 80.
- The key lifetime and identifier are required for a persistent key.

> **Note:**
>
> This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling `psa_get_key_attributes()` with the key's identifier.

`data`

Buffer containing the key data. The content of this buffer is interpreted according to the type declared in `attributes`. All implementations must support at least the format described in the

	documentation of psa_export_key() or psa_export_public_key() for the chosen type. Implementations can support other formats, but be conservative in interpreting the key data: it is recommended that implementations reject content if it might be erroneous, for example, if it is the wrong type or is truncated.
data_length	Size of the data buffer in bytes.
key	On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.

Returns: psa_status_t

PSA_SUCCESS	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_ALREADY_EXISTS	This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.
PSA_ERROR_NOT_SUPPORTED	The key type or key size is not supported, either by the implementation in general or in this particular persistent location.
PSA_ERROR_INVALID_ARGUMENT	The key attributes, as a whole, are invalid.
PSA_ERROR_INVALID_ARGUMENT	The key data is not correctly formatted.
PSA_ERROR_INVALID_ARGUMENT	The size in attributes is nonzero and does not match the size of the key data.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function supports any output from psa_export_key(). Refer to the documentation of psa_export_public_key() for the format of public keys and to the documentation of psa_export_key() for the format for other key types.

The key data determines the key size. The attributes can optionally specify a key size; in this case it must match the size determined from the key data. A key size of 0 in attributes indicates that the key size is solely determined by the key data.

Implementations must reject an attempt to import a key of size 0.

This specification defines a single format for each key type. Implementations can optionally support other formats in addition to the standard format. It is recommended that implementations that support other formats ensure that the formats are clearly unambiguous, to minimize the risk that an invalid input is accidentally interpreted according to a different format.

> **Note:**
>
> The PSA Crypto API does not support asymmetric private key objects outside of a key pair. To import a private key, the `attributes` must specify the corresponding key pair type. Depending on the key type, either the import format contains the public key data or the implementation will reconstruct the public key from the private key as needed.

`psa_generate_key` (function)

Generate a key or key pair.

```
psa_status_t psa_generate_key(const psa_key_attributes_t * attributes,
                              psa_key_id_t * key);
```

Parameters

`attributes` — The attributes for the new key. This function uses the attributes as follows:

- The key type is required. It cannot be an asymmetric public key.
- The key size is required. It must be a valid size for the key type.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see *Permitted algorithms* on page 78.
- The key usage flags define what operations are permitted with the key, see *Key usage flags* on page 80.
- The key lifetime and identifier are required for a persistent key.

> **Note:**
>
> This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling `psa_get_key_attributes()` with the key's identifier.

`key` — On success, an identifier for the newly created key. `PSA_KEY_ID_NULL` on failure.

Returns: `psa_status_t`

`PSA_SUCCESS` — Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.

`PSA_ERROR_ALREADY_EXISTS` — This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.

`PSA_ERROR_NOT_SUPPORTED` — The key type or key size is not supported, either by the implementation in general or in this particular persistent location.

`PSA_ERROR_INVALID_ARGUMENT` — The key attributes, as a whole, are invalid.

PSA_ERROR_INVALID_ARGUMENT	The key type is an asymmetric public key type.
PSA_ERROR_INVALID_ARGUMENT	The key size is not a valid size for the key type.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The key is generated randomly. Its location, policy, type and size are taken from `attributes`.

Implementations must reject an attempt to generate a key of size `0`.

The following type-specific considerations apply:

- For RSA keys (`PSA_KEY_TYPE_RSA_KEY_PAIR`), the public exponent is 65537. The modulus is a product of two probabilistic primes between 2^{n-1} and 2^n where n is the bit size specified in the attributes.

psa_copy_key (function)

Make a copy of a key.

```
psa_status_t psa_copy_key(psa_key_id_t source_key,
                          const psa_key_attributes_t * attributes,
                          psa_key_id_t * target_key);
```

Parameters

`source_key`	The key to copy. It must allow the usage `PSA_KEY_USAGE_COPY`. If a private or secret key is being copied outside of a secure element it must also allow `PSA_KEY_USAGE_EXPORT`.
`attributes`	The attributes for the new key. This function uses the attributes as follows: • The key type and size can be `0`. If either is nonzero, it must match the corresponding attribute of the source key. • The key location (the lifetime and, for persistent keys, the key identifier) is used directly. • The key policy (usage flags and permitted algorithm) are combined from the source key and `attributes` so that both sets

of restrictions apply, as described in the documentation of this function.

> **Note:**
> This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling psa_get_key_attributes() with the key's identifier.

target_key	On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.

Returns: psa_status_t

PSA_SUCCESS	Success. If the new key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_INVALID_HANDLE	source_key is invalid.
PSA_ERROR_ALREADY_EXISTS	This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.
PSA_ERROR_INVALID_ARGUMENT	The lifetime or identifier in attributes are invalid.
PSA_ERROR_INVALID_ARGUMENT	The key policies from source_key and specified in attributes are incompatible.
PSA_ERROR_INVALID_ARGUMENT	attributes specifies a key type or key size which does not match the attributes of source key.
PSA_ERROR_NOT_PERMITTED	source_key does not have the PSA_KEY_USAGE_COPY usage flag.
PSA_ERROR_NOT_PERMITTED	source_key does not have the PSA_KEY_USAGE_EXPORT usage flag and its lifetime does not allow copying it to the target's lifetime.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Copy key material from one location to another.

This function is primarily useful to copy a key from one location to another, as it populates a key using the material from another key which can have a different lifetime.

This function can be used to share a key with a different party, subject to implementation-defined restrictions on key sharing.

The policy on the source key must have the usage flag PSA_KEY_USAGE_COPY set. This flag is sufficient to permit the copy if the key has the lifetime PSA_KEY_LIFETIME_VOLATILE or PSA_KEY_LIFETIME_PERSISTENT. Some secure elements do not provide a way to copy a key without making it extractable from the secure element. If a key is located in such a secure element, then the key must have both usage flags PSA_KEY_USAGE_COPY and PSA_KEY_USAGE_EXPORT in order to make a copy of the key outside the secure element.

The resulting key can only be used in a way that conforms to both the policy of the original key and the policy specified in the `attributes` parameter:

- The usage flags on the resulting key are the bitwise-and of the usage flags on the source policy and the usage flags in `attributes`.
- If both permit the same algorithm or wildcard-based algorithm, the resulting key has the same permitted algorithm.
- If either of the policies permits an algorithm and the other policy allows a wildcard-based permitted algorithm that includes this algorithm, the resulting key uses this permitted algorithm.
- If the policies do not permit any algorithm in common, this function fails with the status PSA_ERROR_INVALID_ARGUMENT.

The effect of this function on implementation-defined attributes is implementation-defined.

9.6.2 Key destruction

psa_destroy_key (function)

Destroy a key.

```
psa_status_t psa_destroy_key(psa_key_id_t key);
```

Parameters

`key`	Identifier of the key to erase. If this is PSA_KEY_ID_NULL, do nothing and return PSA_SUCCESS.

Returns: `psa_status_t`

PSA_SUCCESS	`key` was a valid key identifier and the key material that it referred to has been erased. Alternatively, `key` is PSA_KEY_ID_NULL.
PSA_ERROR_NOT_PERMITTED	The key cannot be erased because it is read-only, either due to a policy or due to physical restrictions.
PSA_ERROR_INVALID_HANDLE	`key` is not a valid handle nor PSA_KEY_ID_NULL.
PSA_ERROR_COMMUNICATION_FAILURE	There was an failure in communication with the cryptoprocessor. The key material might still be present in the cryptoprocessor.
PSA_ERROR_STORAGE_FAILURE	The storage operation failed. Implementations must make a best effort to erase key material even in this situation, however, it might be impossible to guarantee that the key material is not recoverable in such cases.

PSA_ERROR_DATA_CORRUPT	The storage is corrupted. Implementations must make a best effort to erase key material even in this situation, however, it might be impossible to guarantee that the key material is not recoverable in such cases.
PSA_ERROR_DATA_INVALID	
PSA_ERROR_CORRUPTION_DETECTED	An unexpected condition which is not a storage corruption or a communication failure occurred. The cryptoprocessor might have been compromised.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function destroys a key from both volatile memory and, if applicable, non-volatile storage. Implementations must make a best effort to ensure that that the key material cannot be recovered.

This function also erases any metadata such as policies and frees resources associated with the key.

Destroying the key makes the key identifier invalid, and the key identifier must not be used again by the application.

If a key is currently in use in a multi-part operation, then destroying the key will cause the multi-part operation to fail.

psa_purge_key (function)

Remove non-essential copies of key material from memory.

```
psa_status_t psa_purge_key(psa_key_id_t key);
```

Parameters

key	Identifier of the key to purge.

Returns: psa_status_t

PSA_SUCCESS	The key material will have been removed from memory if it is not currently required.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

For keys that have been created with the PSA_KEY_USAGE_CACHE usage flag, an implementation is permitted to make additional copies of the key material that are not in storage and not for the purpose of ongoing operations.

This function will remove these extra copies of the key material from memory.

This function is not required to remove key material from memory in any of the following situations:

- The key is currently in use in a cryptographic operation.
- The key is volatile.

See also *Managing key material* on page 39.

9.6.3 Key export

psa_export_key (function)

Export a key in binary format.

```
psa_status_t psa_export_key(psa_key_id_t key,
                            uint8_t * data,
                            size_t data_size,
                            size_t * data_length);
```

Parameters

key	Identifier of the key to export. It must allow the usage PSA_KEY_USAGE_EXPORT, unless it is a public key.
data	Buffer where the key data is to be written.
data_size	Size of the data buffer in bytes. This must be appropriate for the key: • The required output size is PSA_EXPORT_KEY_OUTPUT_SIZE(type, bits) where type is the key type and bits is the key size in bits. • PSA_EXPORT_KEY_PAIR_MAX_SIZE evaluates to the maximum output size of any supported key pair. • PSA_EXPORT_PUBLIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key. • This API defines no maximum size for symmetric keys. Arbitrarily large data items can be stored in the key store, for example certificates that correspond to a stored private key or input material for key derivation.
data_length	On success, the number of bytes that make up the key data.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_EXPORT flag.
PSA_ERROR_NOT_SUPPORTED	

PSA_ERROR_BUFFER_TOO_SMALL	The size of the `data` buffer is too small. PSA_EXPORT_KEY_OUTPUT_SIZE() or PSA_EXPORT_KEY_PAIR_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The output of this function can be passed to `psa_import_key()` to create an equivalent object.

If the implementation of `psa_import_key()` supports other formats beyond the format specified here, the output from `psa_export_key()` must use the representation specified here, not the original representation.

For standard key types, the output format is as follows:

- For symmetric keys, excluding HMAC keys, the format is the raw bytes of the key.
- For HMAC keys that are shorter than, or equal in size to, the underlying hash algorithm block size, the format is the raw bytes of the key.

  For HMAC keys that are longer than the underlying hash algorithm block size, the format is an IMPLEMENTATION DEFINED choice between the following formats:

  1. The raw bytes of the key.
  2. The raw bytes of the hash of the key, using the underlying hash algorithm.

  See also PSA_KEY_TYPE_HMAC.
- For DES, the key data consists of 8 bytes. The parity bits must be correct.
- For Triple-DES, the format is the concatenation of the two or three DES keys.
- For RSA key pairs, with key type PSA_KEY_TYPE_RSA_KEY_PAIR, the format is the non-encrypted DER encoding of the representation defined by in *PKCS #1: RSA Cryptography Specifications Version 2.2* [RFC8017] as `RSAPrivateKey`, version `0`.

```
RSAPrivateKey ::= SEQUENCE {
    version             INTEGER,  -- must be 0
    modulus             INTEGER,  -- n
    publicExponent      INTEGER,  -- e
    privateExponent     INTEGER,  -- d
    prime1              INTEGER,  -- p
    prime2              INTEGER,  -- q
    exponent1           INTEGER,  -- d mod (p-1)
    exponent2           INTEGER,  -- d mod (q-1)
```

(continues on next page)

(continued from previous page)

```
    coefficient         INTEGER,  -- (inverse of q) mod p
}
```

> **Note:**
> Although it is possible to define an RSA key pair or private key using a subset of these elements, the output from psa_export_key() for an RSA key pair must include all of these elements.

- For elliptic curve key pairs, with key types for which PSA_KEY_TYPE_IS_ECC_KEY_PAIR() is true, the format is a representation of the private value.
  - For Weierstrass curve families PSA_ECC_FAMILY_SECT_XX, PSA_ECC_FAMILY_SECP_XX, PSA_ECC_FAMILY_FRP and PSA_ECC_FAMILY_BRAINPOOL_P_R1, the content of the privateKey field of the ECPrivateKey format defined by *Elliptic Curve Private Key Structure* [RFC5915].
    This is a ceiling(m/8)-byte string in big-endian order where m is the key size in bits.
  - For curve family PSA_ECC_FAMILY_MONTGOMERY, the scalar value of the 'private key' in little-endian order as defined by *Elliptic Curves for Security* [RFC7748] §6. The value must have the forced bits set to zero or one as specified by decodeScalar25519() and decodeScalar448() in [RFC7748] §5.
    This is a ceiling(m/8)-byte string where m is the key size in bits. This is 32 bytes for Curve25519, and 56 bytes for Curve448.
- For Diffie-Hellman key exchange key pairs, with key types for which PSA_KEY_TYPE_IS_DH_KEY_PAIR() is true, the format is the representation of the private key x as a big-endian byte string. The length of the byte string is the private key size in bytes, and leading zeroes are not stripped.
- For public keys, with key types for which PSA_KEY_TYPE_IS_PUBLIC_KEY() is true, the format is the same as for psa_export_public_key().

The policy on the key must have the usage flag PSA_KEY_USAGE_EXPORT set.

psa_export_public_key (function)

Export a public key or the public part of a key pair in binary format.

```
psa_status_t psa_export_public_key(psa_key_id_t key,
                                   uint8_t * data,
                                   size_t data_size,
                                   size_t * data_length);
```

Parameters

key
: Identifier of the key to export.

data
: Buffer where the key data is to be written.

data_size
: Size of the data buffer in bytes. This must be appropriate for the key:
  - The required output size is PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(type, bits) where type is the key type and bits is the key size in bits.

- PSA_EXPORT_PUBLIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key or public part of a key pair.

data_length	On success, the number of bytes that make up the key data.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_INVALID_ARGUMENT	The key is neither a public key nor a key pair.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_BUFFER_TOO_SMALL	The size of the data buffer is too small. PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE() or PSA_EXPORT_PUBLIC_KEY_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The output of this function can be passed to psa_import_key() to create an object that is equivalent to the public key.

If the implementation of psa_import_key() supports other formats beyond the format specified here, the output from psa_export_public_key() must use the representation specified here, not the original representation.

For standard key types, the output format is as follows:

- For RSA public keys, with key type PSA_KEY_TYPE_RSA_PUBLIC_KEY, the DER encoding of the representation defined by *Algorithms and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile* [RFC3279] §2.3.1 as RSAPublicKey.

```
RSAPublicKey ::= SEQUENCE {
   modulus            INTEGER,    -- n
   publicExponent     INTEGER  }  -- e
```

- For elliptic curve key pairs, with key types for which PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY() is true, the format depends on the key family:

- For Weierstrass curve families PSA_ECC_FAMILY_SECT_XX, PSA_ECC_FAMILY_SECP_XX, PSA_ECC_FAMILY_FRP and PSA_ECC_FAMILY_BRAINPOOL_P_R1, the uncompressed representation of an elliptic curve point as an octet string defined in *SEC 1: Elliptic Curve Cryptography* [SEC1] §2.3.3. If `m` is the bit size associated with the curve, i.e. the bit size of `q` for a curve over `F_q`. The representation consists of:
  - The byte `0x04`;
  - `x_P` as a `ceiling(m/8)`-byte string, big-endian;
  - `y_P` as a `ceiling(m/8)`-byte string, big-endian.
- For curve family PSA_ECC_FAMILY_MONTGOMERY, the scalar value of the 'public key' in little-endian order as defined by *Elliptic Curves for Security* [RFC7748] §6. This is a `ceiling(m/8)`-byte string where `m` is the key size in bits.
  - This is 32 bytes for Curve25519, computed as `X25519(private_key, 9)`.
  - This is 56 bytes for Curve448, computed as `X448(private_key, 5)`.

- For Diffie-Hellman key exchange public keys, with key types for which PSA_KEY_TYPE_IS_DH_PUBLIC_KEY is true, the format is the representation of the public key `y = g^x mod p` as a big-endian byte string. The length of the byte string is the length of the base prime `p` in bytes.

Exporting a public key object or the public part of a key pair is always permitted, regardless of the key's usage flags.

PSA_EXPORT_KEY_OUTPUT_SIZE (macro)

Sufficient output buffer size for `psa_export_key()`.

```
#define PSA_EXPORT_KEY_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
```

Parameters

`key_type`	A supported key type.
`key_bits`	The size of the key in bits.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that `psa_export_key()` or `psa_export_public_key()` will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or `0`. If the parameters are not valid, the return value is unspecified.

Description

This macro returns a compile-time constant if its arguments are compile-time constants.

> **Warning:** This function can evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

The following code illustrates how to allocate enough memory to export a key by querying the key type and size at runtime.

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
psa_status_t status;
status = psa_get_key_attributes(key, &attributes);
if (status != PSA_SUCCESS)
    handle_error(...);
psa_key_type_t key_type = psa_get_key_type(&attributes);
size_t key_bits = psa_get_key_bits(&attributes);
size_t buffer_size = PSA_EXPORT_KEY_OUTPUT_SIZE(key_type, key_bits);
psa_reset_key_attributes(&attributes);
uint8_t *buffer = malloc(buffer_size);
if (buffer == NULL)
    handle_error(...);
size_t buffer_length;
status = psa_export_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS)
    handle_error(...);
```

See also `PSA_EXPORT_KEY_PAIR_MAX_SIZE` and `PSA_EXPORT_PUBLIC_KEY_MAX_SIZE`.

PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE (macro)

Sufficient output buffer size for `psa_export_public_key()`.

```
#define PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
```

Parameters

`key_type`	A public key or key pair key type.
`key_bits`	The size of the key in bits.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that `psa_export_public_key()` will not fail with `PSA_ERROR_BUFFER_TOO_SMALL`. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or `0`. If the parameters are not valid, the return value is unspecified.

If the parameters are valid and supported, it is recommended that this macro returns the same result as `PSA_EXPORT_KEY_OUTPUT_SIZE(PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(key_type), key_bits)`.

Description

This macro returns a compile-time constant if its arguments are compile-time constants.

> **Warning:** This function can evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

The following code illustrates how to allocate enough memory to export a public key by querying the key type and size at runtime.

```
psa_key_attributes_t attributes = PSA_KEY_ATTRIBUTES_INIT;
psa_status_t status;
status = psa_get_key_attributes(key, &attributes);
if (status != PSA_SUCCESS)
    handle_error(...);
psa_key_type_t key_type = psa_get_key_type(&attributes);
size_t key_bits = psa_get_key_bits(&attributes);
size_t buffer_size = PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(key_type, key_bits);
psa_reset_key_attributes(&attributes);
uint8_t *buffer = malloc(buffer_size);
if (buffer == NULL)
    handle_error(...);
size_t buffer_length;
status = psa_export_public_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS)
    handle_error(...);
```

See also `PSA_EXPORT_PUBLIC_KEY_MAX_SIZE`.

PSA_EXPORT_KEY_PAIR_MAX_SIZE (macro)

Sufficient buffer size for exporting any asymmetric key pair.

```
#define PSA_EXPORT_KEY_PAIR_MAX_SIZE /* implementation-defined value */
```

This macro must expand to a compile-time constant integer. This value must be a sufficient buffer size when calling `psa_export_key()` to export any asymmetric key pair that is supported by the implementation, regardless of the exact key type and key size.

See also `PSA_EXPORT_KEY_OUTPUT_SIZE()`.

PSA_EXPORT_PUBLIC_KEY_MAX_SIZE (macro)

Sufficient buffer size for exporting any asymmetric public key.

```
#define PSA_EXPORT_PUBLIC_KEY_MAX_SIZE /* implementation-defined value */
```

This macro must expand to a compile-time constant integer. This value must be a sufficient buffer size when calling `psa_export_key()` or `psa_export_public_key()` to export any asymmetric public key that is supported by the implementation, regardless of the exact key type and key size.

See also `PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE()`.

10 Cryptographic operation reference

10.1 Algorithms

This specification encodes algorithms into a structured 32-bit integer value.

Algorithm identifiers are used for two purposes in this API:

1. To specify a specific algorithm to use in a cryptographic operation. These are all defined in *Cryptographic operation reference* on page 98.
2. To specify the policy for a key, identifying the permitted algorithm for use with the key. This use is described in *Key policies* on page 78.

The specific algorithm identifiers are described alongside the cryptographic operation functions to which they apply:

- *Hash algorithms* on page 103
- *MAC algorithms* on page 122
- *Cipher algorithms* on page 135
- *AEAD algorithms* on page 157
- *Key derivation algorithms* on page 181
- *Asymmetric signature algorithms* on page 198
- *Asymmetric encryption algorithms* on page 211
- *Key agreement algorithms* on page 217

10.1.1 Algorithm encoding

`psa_algorithm_t` (type)

Encoding of a cryptographic algorithm.

```
typedef uint32_t psa_algorithm_t;
```

This is a structured bitfield that identifies the category and type of algorithm. The range of algorithm identifier values is divided as follows:

`0x00000000` Reserved as an invalid algorithm identifier.

`0x00000001 - 0x7fffffff`
: Specification-defined algorithm identifiers. Algorithm identifiers defined by this standard always have bit 31 clear. Unallocated algorithm identifier values in this range are reserved for future use.

`0x80000000 - 0xffffffff`
: Implementation-defined algorithm identifiers. Implementations that define additional algorithms must use an encoding with bit 31 set. The related support macros will be easier to write if these algorithm identifier encodings also respect the bitwise structure used by standard encodings.

For algorithms that can be applied to multiple key types, this identifier does not encode the key type. For example, for symmetric ciphers based on a block cipher, `psa_algorithm_t` encodes the block cipher mode and the padding mode while the block cipher itself is encoded via `psa_key_type_t`.

PSA_ALG_NONE (macro)

An invalid algorithm identifier value.

```
#define PSA_ALG_NONE ((psa_algorithm_t)0)
```

Zero is not the encoding of any algorithm.

10.1.2 Algorithm categories

PSA_ALG_IS_HASH (macro)

Whether the specified algorithm is a hash algorithm.

```
#define PSA_ALG_IS_HASH(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a hash algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See *Hash algorithms* on page 103 for a list of defined hash algorithms.

PSA_ALG_IS_MAC (macro)

Whether the specified algorithm is a MAC algorithm.

```
#define PSA_ALG_IS_MAC(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a MAC algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See *MAC algorithms* on page 122 for a list of defined MAC algorithms.

PSA_ALG_IS_CIPHER (macro)

Whether the specified algorithm is a symmetric cipher algorithm.

```
#define PSA_ALG_IS_CIPHER(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

1 if alg is a symmetric cipher algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

Description

See *Cipher algorithms* on page 135 for a list of defined cipher algorithms.

PSA_ALG_IS_AEAD (macro)

Whether the specified algorithm is an authenticated encryption with associated data (AEAD) algorithm.

```
#define PSA_ALG_IS_AEAD(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is an AEAD algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

Description

See *AEAD algorithms* on page 157 for a list of defined AEAD algorithms.

PSA_ALG_IS_SIGN (macro)

Whether the specified algorithm is an asymmetric signature algorithm, also known as public-key signature algorithm.

```
#define PSA_ALG_IS_SIGN(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is an asymmetric signature algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

Description

See *Asymmetric signature algorithms* on page 198 for a list of defined signature algorithms.

PSA_ALG_IS_ASYMMETRIC_ENCRYPTION (macro)

Whether the specified algorithm is an asymmetric encryption algorithm, also known as public-key encryption algorithm.

```
#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) /* specification-defined value */
```

Parameters

`alg` An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is an asymmetric encryption algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See *Asymmetric encryption algorithms* on page 211 for a list of defined asymmetric encryption algorithms.

PSA_ALG_IS_KEY_AGREEMENT (macro)

Whether the specified algorithm is a key agreement algorithm.

```
#define PSA_ALG_IS_KEY_AGREEMENT(alg) /* specification-defined value */
```

Parameters

`alg` An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a key agreement algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See *Key agreement algorithms* on page 217 for a list of defined key agreement algorithms.

PSA_ALG_IS_KEY_DERIVATION (macro)

Whether the specified algorithm is a key derivation algorithm.

```
#define PSA_ALG_IS_KEY_DERIVATION(alg) /* specification-defined value */
```

Parameters

`alg` An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a key derivation algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See *Key derivation algorithms* on page 181 for a list of defined key derivation algorithms.

PSA_ALG_IS_WILDCARD (macro)

Whether the specified algorithm encoding is a wildcard.

```
#define PSA_ALG_IS_WILDCARD(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is a wildcard algorithm encoding.

0 if alg is a non-wildcard algorithm encoding that is suitable for an operation.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

Description

Wildcard algorithm values can only be used to set the permitted algorithm field in a key policy, wildcard values cannot be used to perform an operation.

See PSA_ALG_ANY_HASH for example of how a wildcard algorithm can be used in a key policy.

PSA_ALG_GET_HASH (macro)

Get the hash used by a composite algorithm.

```
#define PSA_ALG_GET_HASH(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

The underlying hash algorithm if alg is a composite algorithm that uses a hash algorithm.

PSA_ALG_NONE if alg is not a composite algorithm that uses a hash.

Description

The following composite algorithms require a hash algorithm:

- PSA_ALG_ECDSA()
- PSA_ALG_HKDF()
- PSA_ALG_HMAC()
- PSA_ALG_RSA_OAEP()
- PSA_ALG_IS_RSA_PKCS1V15_SIGN()
- PSA_ALG_RSA_PSS()
- PSA_ALG_TLS12_PRF()
- PSA_ALG_TLS12_PSK_TO_MS()

10.2 Message digests

10.2.1 Hash algorithms

PSA_ALG_MD2 (macro)

The MD2 message-digest algorithm.

```
#define PSA_ALG_MD2 ((psa_algorithm_t)0x02000001)
```

Warning: The MD2 hash is weak and deprecated and is only recommended for use in legacy protocols.

MD2 is defined in *The MD2 Message-Digest Algorithm* [RFC1319].

PSA_ALG_MD4 (macro)

The MD4 message-digest algorithm.

```
#define PSA_ALG_MD4 ((psa_algorithm_t)0x02000002)
```

Warning: The MD4 hash is weak and deprecated and is only recommended for use in legacy protocols.

MD4 is defined in *The MD4 Message-Digest Algorithm* [RFC1320].

PSA_ALG_MD5 (macro)

The MD5 message-digest algorithm.

```
#define PSA_ALG_MD5 ((psa_algorithm_t)0x02000003)
```

Warning: The MD5 hash is weak and deprecated and is only recommended for use in legacy protocols.

MD5 is defined in *The MD5 Message-Digest Algorithm* [RFC1321].

PSA_ALG_RIPEMD160 (macro)

The RIPEMD-160 message-digest algorithm.

```
#define PSA_ALG_RIPEMD160 ((psa_algorithm_t)0x02000004)
```

RIPEMD-160 is defined in *RIPEMD-160: A Strengthened Version of RIPEMD* [RIPEMD], and also in *ISO/IEC 10118-3:2018 IT Security techniques — Hash-functions — Part 3: Dedicated hash-functions* [ISO10118].

PSA_ALG_SHA_1 (macro)

The SHA-1 message-digest algorithm.

```
#define PSA_ALG_SHA_1 ((psa_algorithm_t)0x02000005)
```

Warning: The SHA-1 hash is weak and deprecated and is only recommended for use in legacy protocols.

SHA-1 is defined in *FIPS Publication 180-4: Secure Hash Standard (SHS)* [FIPS180-4].

PSA_ALG_SHA_224 (macro)

The SHA-224 message-digest algorithm.

```
#define PSA_ALG_SHA_224 ((psa_algorithm_t)0x02000008)
```

SHA-224 is defined in [FIPS180-4].

PSA_ALG_SHA_256 (macro)

The SHA-256 message-digest algorithm.

```
#define PSA_ALG_SHA_256 ((psa_algorithm_t)0x02000009)
```

SHA-256 is defined in [FIPS180-4].

PSA_ALG_SHA_384 (macro)

The SHA-384 message-digest algorithm.

```
#define PSA_ALG_SHA_384 ((psa_algorithm_t)0x0200000a)
```

SHA-384 is defined in [FIPS180-4].

PSA_ALG_SHA_512 (macro)

The SHA-512 message-digest algorithm.

```
#define PSA_ALG_SHA_512 ((psa_algorithm_t)0x0200000b)
```

SHA-512 is defined in [FIPS180-4].

PSA_ALG_SHA_512_224 (macro)

The SHA-512/224 message-digest algorithm.

```
#define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0200000c)
```

SHA-512/224 is defined in [FIPS180-4].

PSA_ALG_SHA_512_256 (macro)

The SHA-512/256 message-digest algorithm.

```
#define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0200000d)
```

SHA-512/256 is defined in [FIPS180-4].

PSA_ALG_SHA3_224 (macro)

The SHA3-224 message-digest algorithm.

```
#define PSA_ALG_SHA3_224 ((psa_algorithm_t)0x02000010)
```

SHA3-224 is defined in *FIPS Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* [FIPS202].

PSA_ALG_SHA3_256 (macro)

The SHA3-256 message-digest algorithm.

```
#define PSA_ALG_SHA3_256 ((psa_algorithm_t)0x02000011)
```

SHA3-256 is defined in [FIPS202].

PSA_ALG_SHA3_384 (macro)

The SHA3-384 message-digest algorithm.

```
#define PSA_ALG_SHA3_384 ((psa_algorithm_t)0x02000012)
```

SHA3-384 is defined in [FIPS202].

PSA_ALG_SHA3_512 (macro)

The SHA3-512 message-digest algorithm.

```
#define PSA_ALG_SHA3_512 ((psa_algorithm_t)0x02000013)
```

SHA3-512 is defined in [FIPS202].

PSA_ALG_SM3 (macro)

The SM3 message-digest algorithm.

```
#define PSA_ALG_SM3 ((psa_algorithm_t)0x02000014)
```

SM3 is defined in *GB/T 32905-2016: Information security techniques — SM3 cryptographic hash algorithm* [PRC-SM3] and *The SM3 Cryptographic Hash Function* [IETF-SM3].

10.2.2 Single-part hashing functions

psa_hash_compute (function)

Calculate the hash (digest) of a message.

```
psa_status_t psa_hash_compute(psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
                              uint8_t * hash,
                              size_t hash_size,
                              size_t * hash_length);
```

Parameters

`alg`	The hash algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(alg)` is true).
`input`	Buffer containing the message to hash.
`input_length`	Size of the `input` buffer in bytes.
`hash`	Buffer where the hash is to be written.
`hash_size`	Size of the `hash` buffer in bytes. This must be at least `PSA_HASH_LENGTH(alg)`.
`hash_length`	On success, the number of bytes that make up the hash value. This is always `PSA_HASH_LENGTH(alg)`.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a hash algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	
`PSA_ERROR_BUFFER_TOO_SMALL`	`hash_size` is too small. `PSA_HASH_LENGTH()` can be used to determine the required buffer size.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

Note:

To verify the hash of a message against an expected value, use `psa_hash_compare()` instead.

`psa_hash_compare` (function)

Calculate the hash (digest) of a message and compare it with a reference value.

```
psa_status_t psa_hash_compare(psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
                              const uint8_t * hash,
                              size_t hash_length);
```

Parameters

`alg`	The hash algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(alg)` is true).
`input`	Buffer containing the message to hash.
`input_length`	Size of the `input` buffer in bytes.
`hash`	Buffer containing the expected hash value.
`hash_length`	Size of the `hash` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	The expected hash is identical to the actual hash of the input.
`PSA_ERROR_INVALID_SIGNATURE`	The hash of the message was calculated successfully, but it differs from the expected hash.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a hash algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`input_length` or `hash_length` do not match the hash size for `alg`
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

10.2.3 Multi-part hashing operations

`psa_hash_operation_t` (type)

The type of the state object for multi-part hash operations.

```
typedef /* implementation-defined type */ psa_hash_operation_t;
```

Before calling any function on a hash operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_hash_operation_t operation;
  memset(&operation, 0, sizeof(operation));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

```
static psa_hash_operation_t operation;
```

- Initialize the object to the initializer `PSA_HASH_OPERATION_INIT`, for example:

```
psa_hash_operation_t operation = PSA_HASH_OPERATION_INIT;
```

- Assign the result of the function `psa_hash_operation_init()` to the object, for example:

```
psa_hash_operation_t operation;
operation = psa_hash_operation_init();
```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

PSA_HASH_OPERATION_INIT (macro)

This macro returns a suitable initializer for a hash operation object of type `psa_hash_operation_t`.

```
#define PSA_HASH_OPERATION_INIT /* implementation-defined value */
```

psa_hash_operation_init (function)

Return an initial value for a hash operation object.

```
psa_hash_operation_t psa_hash_operation_init(void);
```

Returns: `psa_hash_operation_t`

psa_hash_setup (function)

Set up a multi-part hash operation.

```
psa_status_t psa_hash_setup(psa_hash_operation_t * operation,
                            psa_algorithm_t alg);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_hash_operation_t` and not yet in use.
`alg`	The hash algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(alg)` is true).

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not a supported hash algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`alg` is not a hash algorithm.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be inactive.

PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The sequence of operations to calculate a hash (message digest) is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for psa_hash_operation_t, e.g. PSA_HASH_OPERATION_INIT.
3. Call psa_hash_setup() to specify the algorithm.
4. Call psa_hash_update() zero, one or more times, passing a fragment of the message each time. The hash that is calculated is the hash of the concatenation of these messages in order.
5. To calculate the hash, call psa_hash_finish(). To compare the hash with an expected value, call psa_hash_verify(). To suspend the hash operation and extract the current state, call psa_hash_suspend().

If an error occurs at any step after a call to psa_hash_setup(), the operation will need to be reset by a call to psa_hash_abort(). The application can call psa_hash_abort() at any time after the operation has been initialized.

After a successful call to psa_hash_setup(), the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to psa_hash_finish() or psa_hash_verify() or psa_hash_suspend().
- A call to psa_hash_abort().

psa_hash_update (function)

Add a message fragment to a multi-part hash operation.

```
psa_status_t psa_hash_update(psa_hash_operation_t * operation,
                             const uint8_t * input,
                             size_t input_length);
```

Parameters

operation	Active hash operation.
input	Buffer containing the message fragment to hash.
input_length	Size of the input buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call psa_hash_setup() or psa_hash_resume() before calling this function.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_hash_abort().

psa_hash_finish (function)

Finish the calculation of the hash of a message.

```
psa_status_t psa_hash_finish(psa_hash_operation_t * operation,
                             uint8_t * hash,
                             size_t hash_size,
                             size_t * hash_length);
```

Parameters

operation	Active hash operation.
hash	Buffer where the hash is to be written.
hash_size	Size of the hash buffer in bytes. This must be at least PSA_HASH_LENGTH(alg) where alg is the algorithm that the operation performs.
hash_length	On success, the number of bytes that make up the hash value. This is always PSA_HASH_LENGTH(alg) where alg is the hash algorithm that the operation performs.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the hash buffer is too small. PSA_HASH_LENGTH() can be used to determine the required buffer size.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	

PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call psa_hash_setup() or psa_hash_resume() before calling this function. This function calculates the hash of the message formed by concatenating the inputs passed to preceding calls to psa_hash_update().

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling psa_hash_abort().

> **Warning:** It is not recommended to use this function when a specific value is expected for the hash. Call psa_hash_verify() instead with the expected hash value.
>
> Comparing integrity or authenticity data such as hash values with a function such as memcmp() is risky because the time taken by the comparison might leak information about the hashed data which could allow an attacker to guess a valid hash and thereby bypass security controls.

psa_hash_verify (function)

Finish the calculation of the hash of a message and compare it with an expected value.

```
psa_status_t psa_hash_verify(psa_hash_operation_t * operation,
                             const uint8_t * hash,
                             size_t hash_length);
```

Parameters

operation	Active hash operation.
hash	Buffer containing the expected hash value.
hash_length	Size of the hash buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS	The expected hash is identical to the actual hash of the message.
PSA_ERROR_INVALID_SIGNATURE	The hash of the message was calculated successfully, but it differs from the expected hash.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call psa_hash_setup() before calling this function. This function calculates the hash of the message formed by concatenating the inputs passed to preceding calls to psa_hash_update(). It then compares the calculated hash with the expected hash passed as a parameter to this function.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling psa_hash_abort().

> **Note:**
>
> Implementations must make the best effort to ensure that the comparison between the actual hash and the expected hash is performed in constant time.

psa_hash_abort (function)

Abort a hash operation.

```
psa_status_t psa_hash_abort(psa_hash_operation_t * operation);
```

Parameters

operation	Initialized hash operation.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Aborting an operation frees all associated resources except for the operation object itself. Once aborted, the operation object can be reused for another operation by calling psa_hash_setup() again.

This function can be called any time after the operation object has been initialized by one of the methods described in psa_hash_operation_t.

In particular, calling psa_hash_abort() after the operation has been terminated by a call to psa_hash_abort(), psa_hash_finish() or psa_hash_verify() is safe and has no effect.

psa_hash_suspend (function)

Halt the hash operation and extract the intermediate state of the hash computation.

```
psa_status_t psa_hash_suspend(psa_hash_operation_t * operation,
                              uint8_t * hash_state,
                              size_t hash_state_size,
                              size_t * hash_state_length);
```

Parameters

`operation`	Active hash operation.
`hash_state`	Buffer where the hash suspend state is to be written.
`hash_state_size`	Size of the `hash_state` buffer in bytes. This must be appropriate for the selected algorithm: • A sufficient output size is `PSA_HASH_SUSPEND_OUTPUT_SIZE(alg)` where `alg` is the algorithm that was used to set up the operation. • `PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported hash algorithm.
`hash_state_length`	On success, the number of bytes that make up the hash suspend state.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be active.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `hash_state` buffer is too small. `PSA_HASH_SUSPEND_OUTPUT_SIZE()` or `PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE` can be used to determine the required buffer size.
`PSA_ERROR_NOT_SUPPORTED`	The hash algorithm being computed does not support suspend and resume.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call `psa_hash_setup()` or `psa_hash_resume()` before calling this function. This function extracts an intermediate state of the hash computation of the message formed by concatenating the inputs passed to preceding calls to `psa_hash_update()`.

This function can be used to halt a hash operation, and then resume the hash operation at a later time, or in another application, by transferring the extracted hash suspend state to a call to `psa_hash_resume()`.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_hash_abort()`.

Hash suspend and resume is not defined for the SHA3 family of hash algorithms. *Hash suspend state* on page 120 defines the format of the output from `psa_hash_suspend()`.

> **Warning:** Applications must not use any of the hash suspend state as if it was a hash output. Instead, the suspend state must only be used to resume a hash operation, and `psa_hash_finish()` or `psa_hash_verify()` can then calculate or verify the final hash value.

Usage

The sequence of operations to suspend and resume a hash operation is as follows:

1. Compute the first part of the hash.
   a. Allocate an operation object and initialize it as described in the documentation for `psa_hash_operation_t`.
   b. Call `psa_hash_setup()` to specify the algorithm.
   c. Call `psa_hash_update()` zero, one or more times, passing a fragment of the message each time.
   d. Call `psa_hash_suspend()` to extract the hash suspend state into a buffer.
2. Pass the hash state buffer to the application which will resume the operation.
3. Compute the rest of the hash.
   a. Allocate an operation object and initialize it as described in the documentation for `psa_hash_operation_t`.
   b. Call `psa_hash_resume()` with the extracted hash state.
   c. Call `psa_hash_update()` zero, one or more times, passing a fragment of the message each time.
   d. To calculate the hash, call `psa_hash_finish()`. To compare the hash with an expected value, call `psa_hash_verify()`.

If an error occurs at any step after a call to `psa_hash_setup()` or `psa_hash_resume()`, the operation will need to be reset by a call to `psa_hash_abort()`. The application can call `psa_hash_abort()` at any time after the operation has been initialized.

`psa_hash_resume` (function)

Set up a multi-part hash operation using the hash suspend state from a previously suspended hash operation.

```
psa_status_t psa_hash_resume(psa_hash_operation_t * operation,
                             const uint8_t * hash_state,
                             size_t hash_state_length);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_hash_operation_t` and not yet in use.
`hash_state`	A buffer containing the suspended hash state which is to be resumed. This must be in the format output by `psa_hash_suspend()`, which is described in *Hash suspend state format* on page 120.
`hash_state_length`	Length of `hash_state` in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_NOT_SUPPORTED	The provided hash suspend state is for an algorithm that is not supported.
PSA_ERROR_INVALID_ARGUMENT	hash_state does not correspond to a valid hash suspend state. See *Hash suspend state format* on page 120 for the definition.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be inactive.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

See psa_hash_suspend() for an example of how to use this function to suspend and resume a hash operation.

After a successful call to psa_hash_resume(), the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to psa_hash_finish(), psa_hash_verify() or psa_hash_suspend().
- A call to psa_hash_abort().

psa_hash_clone (function)

Clone a hash operation.

```
psa_status_t psa_hash_clone(const psa_hash_operation_t * source_operation,
                            psa_hash_operation_t * target_operation);
```

Parameters

source_operation	The active hash operation to clone.
target_operation	The operation object to set up. It must be initialized but not active.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_BAD_STATE	The source_operation state is not valid: it must be active.
PSA_ERROR_BAD_STATE	The target_operation state is not valid: it must be inactive.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	

PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function copies the state of an ongoing hash operation to a new operation object. In other words, this function is equivalent to calling psa_hash_setup() on target_operation with the same algorithm that source_operation was set up for, then psa_hash_update() on target_operation with the same input that that was passed to source_operation. After this function returns, the two objects are independent, i.e. subsequent calls involving one of the objects do not affect the other object.

10.2.4 Support macros

PSA_HASH_LENGTH (macro)

The size of the output of psa_hash_compute() and psa_hash_finish(), in bytes.

```
#define PSA_HASH_LENGTH(alg) /* implementation-defined value */
```

Parameters

alg	A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true), or an HMAC algorithm (PSA_ALG_HMAC(hash_alg) where hash_alg is a hash algorithm).

Returns

The hash length for the specified hash algorithm. If the hash algorithm is not recognized, return 0. An implementation can return either 0 or the correct size for a hash algorithm that it recognizes, but does not support.

Description

This is also the hash length that psa_hash_compare() and psa_hash_verify() expect.

See also PSA_HASH_MAX_SIZE.

PSA_HASH_MAX_SIZE (macro)

Maximum size of a hash.

```
#define PSA_HASH_MAX_SIZE /* implementation-defined value */
```

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of a hash supported by the implementation, in bytes. The value must not be smaller than this maximum.

See also PSA_HASH_LENGTH().

PSA_HASH_SUSPEND_OUTPUT_SIZE (macro)

A sufficient hash suspend state buffer size for psa_hash_suspend().

```
#define PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) /* specification-defined value */
```

Parameters

alg	A hash algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_HASH(alg) is true).

Returns

A sufficient output size for the algorithm. If the hash algorithm is not recognized, or is not supported by psa_hash_suspend(), return 0. An implementation can return either 0 or a correct size for a hash algorithm that it recognizes, but does not support.

For a supported hash algorithm alg, the following expression is true:

```
PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) == PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH +
                                     PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) +
                                     PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) +
                                     PSA_HASH_BLOCK_LENGTH(alg) - 1
```

Description

If the size of the hash state buffer is at least this large, it is guaranteed that psa_hash_suspend() will not fail due to an insufficient buffer size. The actual size of the output might be smaller in any given call.

See also PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE.

PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE (macro)

A sufficient hash suspend state buffer size for psa_hash_suspend(), for any supported hash algorithms.

```
#define PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE /* implementation-defined value */
```

See also PSA_HASH_SUSPEND_OUTPUT_SIZE().

PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH (macro)

The size of the *algorithm* field that is part of the output of psa_hash_suspend(), in bytes.

```
#define PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH ((size_t)4)
```

Applications can use this value to unpack the hash suspend state that is output by psa_hash_suspend().

PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH (macro)

The size of the *input-length* field that is part of the output of psa_hash_suspend(), in bytes.

```
#define PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) \
    /* specification-defined value */
```

Parameters

`alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(alg)` is true).

Returns

The size, in bytes, of the *input-length* field of the hash suspend state for the specified hash algorithm. If the hash algorithm is not recognized, return `0`. An implementation can return either `0` or the correct size for a hash algorithm that it recognizes, but does not support.

The algorithm-specific values are defined in *Hash suspend state field sizes* on page 121.

Description

Applications can use this value to unpack the hash suspend state that is output by `psa_hash_suspend()`.

PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH (macro)

The size of the *hash-state* field that is part of the output of `psa_hash_suspend()`, in bytes.

```
#define PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) \
    /* specification-defined value */
```

Parameters

`alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(alg)` is true).

Returns

The size, in bytes, of the *hash-state* field of the hash suspend state for the specified hash algorithm. If the hash algorithm is not recognized, return `0`. An implementation can return either `0` or the correct size for a hash algorithm that it recognizes, but does not support.

The algorithm-specific values are defined in *Hash suspend state field sizes* on page 121.

Description

Applications can use this value to unpack the hash suspend state that is output by `psa_hash_suspend()`.

PSA_HASH_BLOCK_LENGTH (macro)

The input block size of a hash algorithm, in bytes.

```
#define PSA_HASH_BLOCK_LENGTH(alg) /* implementation-defined value */
```

Parameters

`alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(alg)` is true).

Returns

The block size in bytes for the specified hash algorithm. If the hash algorithm is not recognized, return `0`. An implementation can return either `0` or the correct size for a hash algorithm that it recognizes, but does not support.

Description

Hash algorithms process their input data in blocks. Hash operations will retain any partial blocks until they have enough input to fill the block or until the operation is finished.

This affects the output from `psa_hash_suspend()`.

10.2.5 Hash suspend state

The hash suspend state is output by `psa_hash_suspend()` and input to `psa_hash_resume()`.

> **Note:**
>
> Hash suspend and resume is not defined for the SHA3 family of hash algorithms.

Hash suspend state format

The hash suspend state has the following format:

hash-suspend-state = algorithm || input-length || hash-state || unprocessed-input

The fields in the hash suspend state are defined as follows:

algorithm
: A big-endian 32-bit unsigned integer.

  The PSA Crypto API algorithm identifier value.

  The byte length of the *algorithm* field can be evaluated using `PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH`.

input-length
: A big-endian unsigned integer

  The content of this field is algorithm-specific:

  - For MD2, this is the number of bytes in the *unprocessed-input*.
  - For all other hash algorithms, this is the total number of bytes of input to the hash computation. This includes the *unprocessed-input* bytes.

  The size of this field is algorithm-specific:

  - For MD2: *input-length* is an 8-bit unsigned integer.
  - For MD4, MD5, RIPEMD-160, SHA-1, SHA-224 and SHA-256: *input-length* is a 64-bit unsigned integer.
  - For SHA-512, SHA-384 and SHA-512/256: *input-length* is a 128-bit unsigned integer.

  The length, in bytes, of the *input-length* field can be calculated using `PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg)` where `alg` is a hash algorithm. See *Hash suspend state field sizes* on page 121.

hash-state
: An array of bytes

  Algorithm-specific intermediate hash state:

  - For MD2: 16 bytes of internal checksum, then 48 bytes of intermediate digest.
  - For MD4 and MD5: 4x 32-bit integers, in little-endian encoding.
  - For RIPEMD-160: 5x 32-bit integers, in little-endian encoding.

- For SHA-1: 5x 32-bit integers, in big-endian encoding.
- For SHA-224 and SHA-256: 8x 32-bit integers, in big-endian encoding.
- For SHA-512, SHA-384 and SHA-512/256: 8x 64-bit integers, in big-endian encoding.

The length of this field is specific to the algorithm. The length, in bytes, of the *hash-state* field can be calculated using `PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg)` where `alg` is a hash algorithm. See *Hash suspend state field sizes*.

unprocessed-input

0 to (*hash-block-size*-1) bytes

A partial block of unprocessed input data. This is between zero and *hash-block-size*-1 bytes of data, the length can be calculated by:

`length(`*unprocessed-input*`)` `=` *input-length* `%` *hash-block-size*.

The *hash-block-size* is specific to the algorithm. The size of a hash block can be calculated using `PSA_HASH_BLOCK_LENGTH(alg)` where `alg` is a hash algorithm. See *Hash suspend state field sizes*.

Hash suspend state field sizes

The following table defines the algorithm-specific field lengths for the hash suspend state returned by `psa_hash_suspend()`. All of the field lengths are in bytes. To compute the field lengths for algorithm `alg`, use the following expressions:

- `PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH` returns the length of the *algorithm* field.
- `PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg)` returns the length of the *input-length* field.
- `PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg)` returns the length of the *hash-state* field.
- `PSA_HASH_BLOCK_LENGTH(alg)-1` is the maximum length of the *unprocessed-bytes* field.
- `PSA_HASH_SUSPEND_OUTPUT_SIZE(slg)` returns the maximum size of the hash suspend state.

Hash algorithm	*input-length* size (bytes)	*hash-state* length (bytes)	*unprocessed-bytes* length (bytes)
`PSA_ALG_MD2`	1	64	0 - 15
`PSA_ALG_MD4`	8	16	0 - 63
`PSA_ALG_MD5`	8	16	0 - 63
`PSA_ALG_RIPEMD160`	8	20	0 - 63
`PSA_ALG_SHA_1`	8	20	0 - 63
`PSA_ALG_SHA_224`	8	32	0 - 63
`PSA_ALG_SHA_256`	8	32	0 - 63
`PSA_ALG_SHA_512_256`	16	64	0 - 127
`PSA_ALG_SHA_384`	16	64	0 - 127
`PSA_ALG_SHA_512`	16	64	0 - 127

10.3 Message authentication codes (MAC)

10.3.1 MAC algorithms

PSA_ALG_HMAC (macro)

Macro to build an HMAC message-authentication-code algorithm from an underlying hash algorithm.

```
#define PSA_ALG_HMAC(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true).

Returns

The corresponding HMAC algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

For example, `PSA_ALG_HMAC(PSA_ALG_SHA_256)` is HMAC-SHA-256.

The HMAC construction is defined in *HMAC: Keyed-Hashing for Message Authentication* [RFC2104].

PSA_ALG_TRUNCATED_MAC (macro)

Macro to build a truncated MAC algorithm.

```
#define PSA_ALG_TRUNCATED_MAC(mac_alg, mac_length) \
    /* specification-defined value */
```

Parameters

`mac_alg`	A MAC algorithm identifier (value of type `psa_algorithm_t` such that `PSA_ALG_IS_MAC(alg)` is true). This can be a truncated or untruncated MAC algorithm.
`mac_length`	Desired length of the truncated MAC in bytes. This must be at most the full length of the MAC and must be at least an implementation-specified minimum. The implementation-specified minimum must not be zero.

Returns

The corresponding MAC algorithm with the specified length.

Unspecified if `alg` is not a supported MAC algorithm or if `mac_length` is too small or too large for the specified MAC algorithm.

Description

A truncated MAC algorithm is identical to the corresponding MAC algorithm except that the MAC value for the truncated algorithm consists of only the first `mac_length` bytes of the MAC value for the untruncated algorithm.

Note:

This macro might allow constructing algorithm identifiers that are not valid, either because the specified length is larger than the untruncated MAC or because the specified length is smaller than permitted by the implementation.

Note:

It is implementation-defined whether a truncated MAC that is truncated to the same length as the MAC of the untruncated algorithm is considered identical to the untruncated algorithm for policy comparison purposes.

The full-length MAC algorithm can be recovered using `PSA_ALG_FULL_LENGTH_MAC()`.

PSA_ALG_CBC_MAC (macro)

The CBC-MAC message-authentication-code algorithm, constructed over a block cipher.

```
#define PSA_ALG_CBC_MAC ((psa_algorithm_t)0x03c00100)
```

> **Warning:** CBC-MAC is insecure in many cases. A more secure mode, such as `PSA_ALG_CMAC`, is recommended.

The CBC-MAC algorithm must be used with a key for a block cipher. For example, one of `PSA_KEY_TYPE_AES`.

CBC-MAC is defined as *MAC Algorithm 1* in *ISO/IEC 9797-1:2011 Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher* [ISO9797].

PSA_ALG_CMAC (macro)

The CMAC message-authentication-code algorithm, constructed over a block cipher.

```
#define PSA_ALG_CMAC ((psa_algorithm_t)0x03c00200)
```

The CMAC algorithm must be used with a key for a block cipher. For example, when used with a key with type `PSA_KEY_TYPE_AES`, the resulting operation is AES-CMAC.

CMAC is defined in *NIST Special Publication 800-38B: Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication* [SP800-38B].

10.3.2 Single-part MAC functions

`psa_mac_compute` (function)

Calculate the message authentication code (MAC) of a message.

```
psa_status_t psa_mac_compute(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const uint8_t * input,
                             size_t input_length,
                             uint8_t * mac,
                             size_t mac_size,
                             size_t * mac_length);
```

Parameters

`key`	Identifier of the key to use for the operation. It must allow the usage `PSA_KEY_USAGE_SIGN_MESSAGE`.
`alg`	The MAC algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_MAC(alg)` is true).
`input`	Buffer containing the input message.
`input_length`	Size of the `input` buffer in bytes.
`mac`	Buffer where the MAC value is to be written.
`mac_size`	Size of the `mac` buffer in bytes. This must be appropriate for the selected algorithm and key: • The exact MAC size is `PSA_MAC_LENGTH(key_type, key_bits, alg)` where `key_type` and `key_bits` are attributes of the key used to compute the MAC. • `PSA_MAC_MAX_SIZE` evaluates to the maximum MAC size of any supported MAC algorithm.
`mac_length`	On success, the number of bytes that make up the MAC value.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_SIGN_MESSAGE` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a MAC algorithm.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `mac` buffer is too small. `PSA_MAC_LENGTH()` or `PSA_MAC_MAX_SIZE` can be used to determine the required buffer size.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	The key could not be retrieved from storage.
`PSA_ERROR_DATA_CORRUPT`	The key could not be retrieved from storage.
`PSA_ERROR_DATA_INVALID`	The key could not be retrieved from storage.

PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

> **Note:**
>
> To verify the MAC of a message against an expected value, use psa_mac_verify() instead. Beware that comparing integrity or authenticity data such as MAC values with a function such as memcmp() is risky because the time taken by the comparison might leak information about the MAC value which could allow an attacker to guess a valid MAC and thereby bypass security controls.

psa_mac_verify (function)

Calculate the MAC of a message and compare it with a reference value.

```
psa_status_t psa_mac_verify(psa_key_id_t key,
                            psa_algorithm_t alg,
                            const uint8_t * input,
                            size_t input_length,
                            const uint8_t * mac,
                            size_t mac_length);
```

Parameters

key	Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_VERIFY_MESSAGE.
alg	The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).
input	Buffer containing the input message.
input_length	Size of the input buffer in bytes.
mac	Buffer containing the expected MAC value.
mac_length	Size of the mac buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS	The expected MAC is identical to the actual MAC of the input.
PSA_ERROR_INVALID_SIGNATURE	The MAC of the message was calculated successfully, but it differs from the expected value.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_VERIFY_MESSAGE flag, or it does not permit the requested algorithm.
PSA_ERROR_INVALID_ARGUMENT	key is not compatible with alg.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	

PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	The key could not be retrieved from storage.
PSA_ERROR_DATA_CORRUPT	The key could not be retrieved from storage.
PSA_ERROR_DATA_INVALID	The key could not be retrieved from storage.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

10.3.3 Multi-part MAC operations

psa_mac_operation_t (type)

The type of the state object for multi-part MAC operations.

```
typedef /* implementation-defined type */ psa_mac_operation_t;
```

Before calling any function on a MAC operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_mac_operation_t operation;
  memset(&operation, 0, sizeof(operation));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

  ```
  static psa_mac_operation_t operation;
  ```

- Initialize the object to the initializer PSA_MAC_OPERATION_INIT, for example:

  ```
  psa_mac_operation_t operation = PSA_MAC_OPERATION_INIT;
  ```

- Assign the result of the function psa_mac_operation_init() to the object, for example:

  ```
  psa_mac_operation_t operation;
  operation = psa_mac_operation_init();
  ```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

PSA_MAC_OPERATION_INIT (macro)

This macro returns a suitable initializer for a MAC operation object of type psa_mac_operation_t.

```
#define PSA_MAC_OPERATION_INIT /* implementation-defined value */
```

psa_mac_operation_init (function)

Return an initial value for a MAC operation object.

```
psa_mac_operation_t psa_mac_operation_init(void);
```

Returns: psa_mac_operation_t

psa_mac_sign_setup (function)

Set up a multi-part MAC calculation operation.

```
psa_status_t psa_mac_sign_setup(psa_mac_operation_t * operation,
                                psa_key_id_t key,
                                psa_algorithm_t alg);
```

Parameters

operation	The operation object to set up. It must have been initialized as per the documentation for psa_mac_operation_t and not yet in use.
key	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_SIGN_MESSAGE.
alg	The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_SIGN_MESSAGE flag, or it does not permit the requested algorithm.
PSA_ERROR_INVALID_ARGUMENT	key is not compatible with alg.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	The key could not be retrieved from storage.
PSA_ERROR_DATA_CORRUPT	The key could not be retrieved from storage.
PSA_ERROR_DATA_INVALID	The key could not be retrieved from storage.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be inactive.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function sets up the calculation of the message authentication code (MAC) of a byte string. To verify the MAC of a message against an expected value, use psa_mac_verify_setup() instead.

The sequence of operations to calculate a MAC is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for psa_mac_operation_t, e.g. PSA_MAC_OPERATION_INIT.
3. Call psa_mac_sign_setup() to specify the algorithm and key.
4. Call psa_mac_update() zero, one or more times, passing a fragment of the message each time. The MAC that is calculated is the MAC of the concatenation of these messages in order.
5. At the end of the message, call psa_mac_sign_finish() to finish calculating the MAC value and retrieve it.

If an error occurs at any step after a call to psa_mac_sign_setup(), the operation will need to be reset by a call to psa_mac_abort(). The application can call psa_mac_abort() at any time after the operation has been initialized.

After a successful call to psa_mac_sign_setup(), the application must eventually terminate the operation through one of the following methods:

- A successful call to psa_mac_sign_finish().
- A call to psa_mac_abort().

psa_mac_verify_setup (function)

Set up a multi-part MAC verification operation.

```
psa_status_t psa_mac_verify_setup(psa_mac_operation_t * operation,
                                  psa_key_id_t key,
                                  psa_algorithm_t alg);
```

Parameters

operation	The operation object to set up. It must have been initialized as per the documentation for psa_mac_operation_t and not yet in use.
key	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_VERIFY_MESSAGE.
alg	The MAC algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_VERIFY_MESSAGE flag, or it does not permit the requested algorithm.

PSA_ERROR_INVALID_ARGUMENT	key is not compatible with alg.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not a MAC algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	The key could not be retrieved from storage
PSA_ERROR_DATA_CORRUPT	The key could not be retrieved from storage.
PSA_ERROR_DATA_INVALID	The key could not be retrieved from storage.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be inactive.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function sets up the verification of the message authentication code (MAC) of a byte string against an expected value.

The sequence of operations to verify a MAC is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for psa_mac_operation_t, e.g. PSA_MAC_OPERATION_INIT.
3. Call psa_mac_verify_setup() to specify the algorithm and key.
4. Call psa_mac_update() zero, one or more times, passing a fragment of the message each time. The MAC that is calculated is the MAC of the concatenation of these messages in order.
5. At the end of the message, call psa_mac_verify_finish() to finish calculating the actual MAC of the message and verify it against the expected value.

If an error occurs at any step after a call to psa_mac_verify_setup(), the operation will need to be reset by a call to psa_mac_abort(). The application can call psa_mac_abort() at any time after the operation has been initialized.

After a successful call to psa_mac_verify_setup(), the application must eventually terminate the operation through one of the following methods:

- A successful call to psa_mac_verify_finish().
- A call to psa_mac_abort().

psa_mac_update (function)

Add a message fragment to a multi-part MAC operation.

```
psa_status_t psa_mac_update(psa_mac_operation_t * operation,
                            const uint8_t * input,
                            size_t input_length);
```

Parameters

`operation`	Active MAC operation.
`input`	Buffer containing the message fragment to add to the MAC calculation.
`input_length`	Size of the `input` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be active.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call `psa_mac_sign_setup()` or `psa_mac_verify_setup()` before calling this function.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_mac_abort()`.

`psa_mac_sign_finish` (function)

Finish the calculation of the MAC of a message.

```
psa_status_t psa_mac_sign_finish(psa_mac_operation_t * operation,
                                 uint8_t * mac,
                                 size_t mac_size,
                                 size_t * mac_length);
```

Parameters

`operation`	Active MAC operation.
`mac`	Buffer where the MAC value is to be written.
`mac_size`	Size of the `mac` buffer in bytes. This must be appropriate for the selected algorithm and key:

- The exact MAC size is PSA_MAC_LENGTH(key_type, key_bits, alg) where key_type and key_bits are attributes of the key, and alg is the algorithm used to compute the MAC.
- PSA_MAC_MAX_SIZE evaluates to the maximum MAC size of any supported MAC algorithm.

mac_length	On success, the number of bytes that make up the MAC value. This is always PSA_MAC_FINAL_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of the key and alg is the MAC algorithm that is calculated.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be an active mac sign operation.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the mac buffer is too small. PSA_MAC_LENGTH() or PSA_MAC_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call psa_mac_sign_setup() before calling this function. This function calculates the MAC of the message formed by concatenating the inputs passed to preceding calls to psa_mac_update().

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling psa_mac_abort().

> **Warning:** It is not recommended to use this function when a specific value is expected for the MAC. Call psa_mac_verify_finish() instead with the expected MAC value.
>
> Comparing integrity or authenticity data such as MAC values with a function such as memcmp() is risky because the time taken by the comparison might leak information about the hashed data which could allow an attacker to guess a valid MAC and thereby bypass security controls.

psa_mac_verify_finish (function)

Finish the calculation of the MAC of a message and compare it with an expected value.

```
psa_status_t psa_mac_verify_finish(psa_mac_operation_t * operation,
                                   const uint8_t * mac,
                                   size_t mac_length);
```

Parameters

`operation`	Active MAC operation.
`mac`	Buffer containing the expected MAC value.
`mac_length`	Size of the `mac` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	The expected MAC is identical to the actual MAC of the message.
`PSA_ERROR_INVALID_SIGNATURE`	The MAC of the message was calculated successfully, but it differs from the expected MAC.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be an active mac verify operation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call `psa_mac_verify_setup()` before calling this function. This function calculates the MAC of the message formed by concatenating the inputs passed to preceding calls to `psa_mac_update()`. It then compares the calculated MAC with the expected MAC passed as a parameter to this function.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_mac_abort()`.

> **Note:**
>
> Implementations must make the best effort to ensure that the comparison between the actual MAC and the expected MAC is performed in constant time.

`psa_mac_abort` (function)

Abort a MAC operation.

```
psa_status_t psa_mac_abort(psa_mac_operation_t * operation);
```

Parameters

`operation`	Initialized MAC operation.

Returns: `psa_status_t`

`PSA_SUCCESS`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

Aborting an operation frees all associated resources except for the `operation` object itself. Once aborted, the operation object can be reused for another operation by calling `psa_mac_sign_setup()` or `psa_mac_verify_setup()` again.

This function can be called any time after the operation object has been initialized by one of the methods described in `psa_mac_operation_t`.

In particular, calling `psa_mac_abort()` after the operation has been terminated by a call to `psa_mac_abort()`, `psa_mac_sign_finish()` or `psa_mac_verify_finish()` is safe and has no effect.

10.3.4 Support macros

PSA_ALG_IS_HMAC (macro)

Whether the specified algorithm is an HMAC algorithm.

```
#define PSA_ALG_IS_HMAC(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is an HMAC algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

HMAC is a family of MAC algorithms that are based on a hash function.

PSA_ALG_IS_BLOCK_CIPHER_MAC (macro)

Whether the specified algorithm is a MAC algorithm based on a block cipher.

```
#define PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is a MAC algorithm based on a block cipher, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

PSA_ALG_FULL_LENGTH_MAC (macro)

Macro to construct the MAC algorithm with a full length MAC, from a truncated MAC algorithm.

```
#define PSA_ALG_FULL_LENGTH_MAC(mac_alg) /* specification-defined value */
```

Parameters

mac_alg	A MAC algorithm identifier (value of type psa_algorithm_t such that PSA_ALG_IS_MAC(alg) is true). This can be a truncated or untruncated MAC algorithm.

Returns

The corresponding MAC algorithm with a full length MAC.

Unspecified if alg is not a supported MAC algorithm.

PSA_MAC_LENGTH (macro)

The size of the output of psa_mac_compute() and psa_mac_sign_finish(), in bytes.

```
#define PSA_MAC_LENGTH(key_type, key_bits, alg) \
    /* implementation-defined value */
```

Parameters

key_type	The type of the MAC key.
key_bits	The size of the MAC key in bits.
alg	A MAC algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_MAC(alg) is true).

Returns

The MAC length for the specified algorithm with the specified key parameters.

0 if the MAC algorithm is not recognized.

Either 0 or the correct length for a MAC algorithm that the implementation recognizes, but does not support.

Unspecified if the key parameters are not consistent with the algorithm.

Description

This is also the MAC length that psa_mac_verify() and psa_mac_verify_finish() expects.

See also PSA_MAC_MAX_SIZE.

PSA_MAC_MAX_SIZE (macro)

Maximum size of a MAC.

```
#define PSA_MAC_MAX_SIZE /* implementation-defined value */
```

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of a MAC supported by the implementation, in bytes. The value must not be smaller than this maximum.

See also PSA_MAC_LENGTH().

10.4 Unauthenticated ciphers

> **Warning:** The unauthenticated cipher API is provided to implement legacy protocols and for use cases where the data integrity and authenticity is guaranteed by non-cryptographic means.
>
> It is recommended that newer protocols use *Authenticated encryption with associated data (AEAD)* on page 157.

10.4.1 Cipher algorithms

PSA_ALG_STREAM_CIPHER (macro)

The stream cipher mode of a stream cipher algorithm.

```
#define PSA_ALG_STREAM_CIPHER ((psa_algorithm_t)0x04800100)
```

The underlying stream cipher is determined by the key type. The ARC4 and ChaCha20 ciphers use this algorithm identifier.

ARC4

To use ARC4, use a key type of PSA_KEY_TYPE_ARC4 and algorithm id PSA_ALG_STREAM_CIPHER.

> **Warning:** The ARC4 cipher is weak and deprecated and is only recommended for use in legacy protocols.

The ARC4 cipher does not use an initialization vector (IV). When using a multi-part cipher operation with the PSA_ALG_STREAM_CIPHER algorithm and an ARC4 key, psa_cipher_generate_iv() and psa_cipher_set_iv() must not be called.

ChaCha20

To use ChaCha20, use a key type of PSA_KEY_TYPE_CHACHA20 and algorithm id PSA_ALG_STREAM_CIPHER.

Implementations must support the variant that is defined in *ChaCha20 and Poly1305 for IETF Protocols* [RFC7539] §2.4, which has a 96-bit nonce and a 32-bit counter. Implementations can optionally also support the original variant, as defined in *ChaCha, a variant of Salsa20* [CHACHA20], which has a 64-bit nonce and a 64-bit counter. Except where noted, the [RFC7539] variant must be used.

ChaCha20 defines a nonce and an initial counter to be provided to the encryption and decryption operations. When using a ChaCha20 key with the PSA_ALG_STREAM_CIPHER algorithm, these values are provided using the initialization vector (IV) functions in the following ways:

- A call to psa_cipher_encrypt() will generate a random 12-byte nonce, and set the counter value to zero. The random nonce is output as a 12-byte IV value in the output.
- A call to psa_cipher_decrypt() will use first 12 bytes of the input buffer as the nonce and set the counter value to zero.
- A call to psa_cipher_generate_iv() on a multi-part cipher operation will generate and return a random 12-byte nonce and set the counter value to zero.
- A call to psa_cipher_set_iv() on a multi-part cipher operation can support the following IV sizes:
  - 12 bytes: the provided IV is used as the nonce, and the counter value is set to zero.
  - 16 bytes: the first four bytes of the IV are used as the counter value (encoded as little-endian), and the remaining 12 bytes is used as the nonce.
  - 8 bytes: the cipher operation uses the original [CHACHA20] definition of ChaCha20: the provided IV is used as the 64-bit nonce, and the 64-bit counter value is set to zero.
  - It is recommended that implementations do not support other sizes of IV.

PSA_ALG_CTR (macro)

A stream cipher built using the Counter (CTR) mode of a block cipher.

```
#define PSA_ALG_CTR ((psa_algorithm_t)0x04c01000)
```

CTR is a stream cipher which is built from a block cipher. The underlying block cipher is determined by the key type. For example, to use AES-128-CTR, use this algorithm with a key of type PSA_KEY_TYPE_AES and a size of 128 bits (16 bytes).

The CTR block cipher mode is defined in *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [SP800-38A].

CTR mode requires a *counter block* which is the same size as the cipher block length. The counter block is updated for each block (or a partial final block) that is encrypted or decrypted.

A counter block value must only be used once across all messages encrypted using the same key value. This is typically achieved by splitting the counter block into a nonce, which is unique among all message encrypted with the key, and a counter which is incremented for each block of a message.

For example, when using AES-CTR encryption, which uses a 16-byte block, the application can provide a 12-byte nonce when setting the IV. This leaves 4 bytes for the counter, allowing up to 2^32 blocks (64GB) of message data to be encrypted in each message.

The first counter block is constructed from the initialization vector (IV). The initial counter block is is constructed in the following ways:

- A call to `psa_cipher_encrypt()` will generate a random counter block value. This is the first block of output.
- A call to `psa_cipher_decrypt()` will use first block of the input buffer as the initial counter block value.
- A call to `psa_cipher_generate_iv()` on a multi-part cipher operation will generate and return a random counter block value.
- A call to `psa_cipher_set_iv()` on a multi-part cipher operation requires an IV must be between `1` and *n* bytes in length, where *n* is the cipher block length. The counter block is initialized using the IV, and padded with zero bytes up to the block length.

During the counter block update operation, the counter block is treated as a single big-endian encoded integer and the update operation increments this integer by `1`.

This scheme meets the recommendations in Appendix B of [SP800-38A].

> **Note:**
>
> The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

PSA_ALG_CFB (macro)

A stream cipher built using the Cipher Feedback (CFB) mode of a block cipher.

```
#define PSA_ALG_CFB ((psa_algorithm_t)0x04c01100)
```

The underlying block cipher is determined by the key type. This is the variant of CFB where each iteration encrypts or decrypts a segment of the input that is the same length as the cipher block size. For example, using `PSA_ALG_CFB` with a key of type `PSA_KEY_TYPE_AES` will result in the AES-CFB-128 cipher.

CFB mode requires an initialization vector (IV) that is the same size as the cipher block length.

> **Note:**
>
> The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

The CFB block cipher mode is defined in *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [SP800-38A], using a segment size *s* equal to the block size *b*. The definition in [SP800-38A] is extended to allow an incomplete final block of input, in which case the algorithm discards the final bytes of the key stream when encrypting or decrypting the final partial block.

PSA_ALG_OFB (macro)

A stream cipher built using the Output Feedback (OFB) mode of a block cipher.

```
#define PSA_ALG_OFB ((psa_algorithm_t)0x04c01200)
```

The underlying block cipher is determined by the key type.

OFB mode requires an initialization vector (IV) that is the same size as the cipher block length. OFB mode requires that the IV is a nonce, and must be unique for each use of the mode with the same key.

> **Note:**
>
> The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

The OFB block cipher mode is defined in *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [SP800-38A].

PSA_ALG_XTS (macro)

The XEX with Ciphertext Stealing (XTS) cipher mode of a block cipher.

```
#define PSA_ALG_XTS ((psa_algorithm_t)0x0440ff00)
```

XTS is a cipher mode which is built from a block cipher, designed for use in disk encryption. It requires at least one full cipher block length of input, but beyond this minimum the input does not need to be a whole number of blocks.

XTS mode uses two keys for the underlying block cipher. These are provided by using a key that is twice the normal key size for the cipher. For example, to use AES-256-XTS the application must create a key with type `PSA_KEY_TYPE_AES` and bit size `512`.

XTS mode requires an initialization vector (IV) that is the same size as the cipher block length. The IV for XTS is typically defined to be the sector number of the disk block being encrypted or decrypted.

The XTS block cipher mode is defined in *1619-2018 - IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices* [IEEE-XTS].

`PSA_ALG_ECB_NO_PADDING` (macro)

The Electronic Codebook (ECB) mode of a block cipher, with no padding.

```
#define PSA_ALG_ECB_NO_PADDING ((psa_algorithm_t)0x04404400)
```

> **Warning:** ECB mode does not protect the confidentiality of the encrypted data except in extremely narrow circumstances. It is recommended that applications only use ECB if they need to construct an operating mode that the implementation does not provide. Implementations are encouraged to provide the modes that applications need in preference to supporting direct access to ECB.

The underlying block cipher is determined by the key type.

This symmetric cipher mode can only be used with messages whose lengths are a multiple of the block size of the chosen block cipher.

ECB mode does not accept an initialization vector (IV). When using a multi-part cipher operation with this algorithm, `psa_cipher_generate_iv()` and `psa_cipher_set_iv()` must not be called.

Note:

The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

The ECB block cipher mode is defined in *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [SP800-38A].

`PSA_ALG_CBC_NO_PADDING` (macro)

The Cipher Block Chaining (CBC) mode of a block cipher, with no padding.

```
#define PSA_ALG_CBC_NO_PADDING ((psa_algorithm_t)0x04404000)
```

The underlying block cipher is determined by the key type.

This symmetric cipher mode can only be used with messages whose lengths are a multiple of the block size of the chosen block cipher.

CBC mode requires an initialization vector (IV) that is the same size as the cipher block length.

Note:

The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

The CBC block cipher mode is defined in *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [SP800-38A].

`PSA_ALG_CBC_PKCS7` (macro)

The Cipher Block Chaining (CBC) mode of a block cipher, with PKCS#7 padding.

```
#define PSA_ALG_CBC_PKCS7 ((psa_algorithm_t)0x04404100)
```

The underlying block cipher is determined by the key type.

CBC mode requires an initialization vector (IV) that is the same size as the cipher block length.

Note:

The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

The CBC block cipher mode is defined in *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques* [SP800-38A]. The padding operation is defined by *PKCS #7: Cryptographic Message Syntax Version 1.5* [RFC2315] §10.3.

10.4.2 Single-part cipher functions

`psa_cipher_encrypt` (function)

Encrypt a message using a symmetric cipher.

```
psa_status_t psa_cipher_encrypt(psa_key_id_t key,
                                psa_algorithm_t alg,
                                const uint8_t * input,
                                size_t input_length,
                                uint8_t * output,
                                size_t output_size,
                                size_t * output_length);
```

Parameters

`key`	Identifier of the key to use for the operation. It must allow the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The cipher algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_CIPHER(alg)` is true).
`input`	Buffer containing the message to encrypt.
`input_length`	Size of the `input` buffer in bytes.
`output`	Buffer where the output is to be written. The output contains the IV followed by the ciphertext proper.
`output_size`	Size of the `output` buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is `PSA_CIPHER_ENCRYPT_OUTPUT_SIZE(key_type, alg, input_length)` where `key_type` is the type of `key`. • `PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE(input_length)` evaluates to the maximum output size of any supported cipher encryption.
`output_length`	On success, the number of bytes that make up the output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_ENCRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`key` is not compatible with `alg`.
`PSA_ERROR_INVALID_ARGUMENT`	The `input_length` is not valid for the algorithm and key type. For example, the algorithm is a based on block cipher and requires a whole number of blocks, but the total input size is not a multiple of the block size.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a cipher algorithm.
`PSA_ERROR_BUFFER_TOO_SMALL`	`output_size` is too small. `PSA_CIPHER_ENCRYPT_OUTPUT_SIZE()` or `PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE()` can be used to determine the required buffer size.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	

PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function encrypts a message with a random initialization vector (IV). The length of the IV is PSA_CIPHER_IV_LENGTH(key_type, alg) where key_type is the type of key. The output of psa_cipher_encrypt() is the IV followed by the ciphertext.

Use the multi-part operation interface with a psa_cipher_operation_t object to provide other forms of IV or to manage the IV and ciphertext independently.

psa_cipher_decrypt (function)

Decrypt a message using a symmetric cipher.

```
psa_status_t psa_cipher_decrypt(psa_key_id_t key,
                                psa_algorithm_t alg,
                                const uint8_t * input,
                                size_t input_length,
                                uint8_t * output,
                                size_t output_size,
                                size_t * output_length);
```

Parameters

key	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_DECRYPT.
alg	The cipher algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).
input	Buffer containing the message to decrypt. This consists of the IV followed by the ciphertext proper.
input_length	Size of the input buffer in bytes.
output	Buffer where the plaintext is to be written.
output_size	Size of the output buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is PSA_CIPHER_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length) where key_type is the type of key. • PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported cipher decryption.

`output_length`	On success, the number of bytes that make up the output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_DECRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`key` is not compatible with `alg`.
`PSA_ERROR_INVALID_ARGUMENT`	The `input_length` is not valid for the algorithm and key type. For example, the algorithm is a based on block cipher and requires a whole number of blocks, but the total input size is not a multiple of the block size.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a cipher algorithm.
`PSA_ERROR_BUFFER_TOO_SMALL`	`output_size` is too small. `PSA_CIPHER_DECRYPT_OUTPUT_SIZE()` or `PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE()` can be used to determine the required buffer size.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function decrypts a message encrypted with a symmetric cipher.

The input to this function must contain the IV followed by the ciphertext, as output by `psa_cipher_encrypt()`. The IV must be `PSA_CIPHER_IV_LENGTH(key_type, alg)` bytes in length, where `key_type` is the type of `key`.

Use the multi-part operation interface with a `psa_cipher_operation_t` object to decrypt data which is not in the expected input format.

10.4.3 Multi-part cipher operations

`psa_cipher_operation_t` (type)

The type of the state object for multi-part cipher operations.

```
typedef /* implementation-defined type */ psa_cipher_operation_t;
```

Before calling any function on a cipher operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

```
psa_cipher_operation_t operation;
memset(&operation, 0, sizeof(operation));
```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

```
static psa_cipher_operation_t operation;
```

- Initialize the object to the initializer `PSA_CIPHER_OPERATION_INIT`, for example:

```
psa_cipher_operation_t operation = PSA_CIPHER_OPERATION_INIT;
```

- Assign the result of the function `psa_cipher_operation_init()` to the object, for example:

```
psa_cipher_operation_t operation;
operation = psa_cipher_operation_init();
```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

`PSA_CIPHER_OPERATION_INIT` (macro)

This macro returns a suitable initializer for a cipher operation object of type `psa_cipher_operation_t`.

```
#define PSA_CIPHER_OPERATION_INIT /* implementation-defined value */
```

`psa_cipher_operation_init` (function)

Return an initial value for a cipher operation object.

```
psa_cipher_operation_t psa_cipher_operation_init(void);
```

Returns: `psa_cipher_operation_t`

`psa_cipher_encrypt_setup` (function)

Set the key for a multi-part symmetric encryption operation.

```
psa_status_t psa_cipher_encrypt_setup(psa_cipher_operation_t * operation,
                                      psa_key_id_t key,
                                      psa_algorithm_t alg);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_cipher_operation_t` and not yet in use.

`key`	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The cipher algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_CIPHER(alg)` is true).

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_ENCRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a cipher algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be inactive.
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The sequence of operations to encrypt a message with a symmetric cipher is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for `psa_cipher_operation_t`, e.g. `PSA_CIPHER_OPERATION_INIT`.
3. Call `psa_cipher_encrypt_setup()` to specify the algorithm and key.
4. Call either `psa_cipher_generate_iv()` or `psa_cipher_set_iv()` to generate or set the initialization vector (IV), if the algorithm requires one. It is recommended to use `psa_cipher_generate_iv()` unless the protocol being implemented requires a specific IV value.
5. Call `psa_cipher_update()` zero, one or more times, passing a fragment of the message each time.
6. Call `psa_cipher_finish()`.

If an error occurs at any step after a call to `psa_cipher_encrypt_setup()`, the operation will need to be reset by a call to `psa_cipher_abort()`. The application can call `psa_cipher_abort()` at any time after the operation has been initialized.

After a successful call to psa_cipher_encrypt_setup(), the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to psa_cipher_finish().
- A call to psa_cipher_abort().

psa_cipher_decrypt_setup (function)

Set the key for a multi-part symmetric decryption operation.

```
psa_status_t psa_cipher_decrypt_setup(psa_cipher_operation_t * operation,
                                      psa_key_id_t key,
                                      psa_algorithm_t alg);
```

Parameters

operation	The operation object to set up. It must have been initialized as per the documentation for psa_cipher_operation_t and not yet in use.
key	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage PSA_KEY_USAGE_DECRYPT.
alg	The cipher algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
PSA_ERROR_INVALID_ARGUMENT	key is not compatible with alg.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not a cipher algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be inactive.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The sequence of operations to decrypt a message with a symmetric cipher is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for `psa_cipher_operation_t`, e.g. `PSA_CIPHER_OPERATION_INIT`.
3. Call `psa_cipher_decrypt_setup()` to specify the algorithm and key.
4. Call `psa_cipher_set_iv()` with the initialization vector (IV) for the decryption, if the algorithm requires one. This must match the IV used for the encryption.
5. Call `psa_cipher_update()` zero, one or more times, passing a fragment of the message each time.
6. Call `psa_cipher_finish()`.

If an error occurs at any step after a call to `psa_cipher_decrypt_setup()`, the operation will need to be reset by a call to `psa_cipher_abort()`. The application can call `psa_cipher_abort()` at any time after the operation has been initialized.

After a successful call to `psa_cipher_decrypt_setup()`, the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to `psa_cipher_finish()`.
- A call to `psa_cipher_abort()`.

`psa_cipher_generate_iv` (function)

Generate an initialization vector (IV) for a symmetric encryption operation.

```
psa_status_t psa_cipher_generate_iv(psa_cipher_operation_t * operation,
                                    uint8_t * iv,
                                    size_t iv_size,
                                    size_t * iv_length);
```

Parameters

`operation`	Active cipher operation.
`iv`	Buffer where the generated IV is to be written.
`iv_size`	Size of the `iv` buffer in bytes. This must be at least `PSA_CIPHER_IV_LENGTH(key_type, alg)` where `key_type` and `alg` are type of key and the algorithm respectively that were used to set up the cipher operation.
`iv_length`	On success, the number of bytes of the generated IV.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	Either: • The cipher algorithm does not use an IV. • The operation state is not valid: it must be active, with no IV set.

PSA_ERROR_BUFFER_TOO_SMALL	The size of the iv buffer is too small. PSA_CIPHER_IV_LENGTH() or PSA_CIPHER_IV_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function generates a random IV, nonce or initial counter value for the encryption operation as appropriate for the chosen algorithm, key type and key size.

The generated IV is always the default length for the key and algorithm: PSA_CIPHER_IV_LENGTH(key_type, alg), where key_type is the type of key and alg is the algorithm that were used to set up the operation. To generate different lengths of IV, use psa_generate_random() and psa_cipher_set_iv().

If the cipher algorithm does not use an IV, calling this function returns a PSA_ERROR_BAD_STATE error. For these algorithms, PSA_CIPHER_IV_LENGTH(key_type, alg) will be zero.

The application must call psa_cipher_encrypt_setup() before calling this function.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_cipher_abort().

psa_cipher_set_iv (function)

Set the initialization vector (IV) for a symmetric encryption or decryption operation.

```
psa_status_t psa_cipher_set_iv(psa_cipher_operation_t * operation,
                               const uint8_t * iv,
                               size_t iv_length);
```

Parameters

operation	Active cipher operation.
iv	Buffer containing the IV to use.
iv_length	Size of the IV in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	Either: • The cipher algorithm does not use an IV.

	• The operation state is not valid: it must be an active cipher encrypt operation, with no IV set.
PSA_ERROR_INVALID_ARGUMENT	The size of iv is not acceptable for the chosen algorithm, or the chosen algorithm does not use an IV.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function sets the IV, nonce or initial counter value for the encryption or decryption operation.

If the cipher algorithm does not use an IV, calling this function returns a PSA_ERROR_BAD_STATE error. For these algorithms, PSA_CIPHER_IV_LENGTH(key_type, alg) will be zero.

The application must call psa_cipher_encrypt_setup() or psa_cipher_decrypt_setup() before calling this function.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_cipher_abort().

> **Note:**
>
> When encrypting, psa_cipher_generate_iv() is recommended instead of using this function, unless implementing a protocol that requires a non-random IV.

psa_cipher_update (function)

Encrypt or decrypt a message fragment in an active cipher operation.

```
psa_status_t psa_cipher_update(psa_cipher_operation_t * operation,
                               const uint8_t * input,
                               size_t input_length,
                               uint8_t * output,
                               size_t output_size,
                               size_t * output_length);
```

Parameters

operation	Active cipher operation.
input	Buffer containing the message fragment to encrypt or decrypt.

`input_length`	Size of the `input` buffer in bytes.
`output`	Buffer where the output is to be written.
`output_size`	Size of the `output` buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is `PSA_CIPHER_UPDATE_OUTPUT_SIZE(key_type, alg, input_length)` where `key_type` is the type of key and `alg` is the algorithm that were used to set up the operation. • `PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE(input_length)` evaluates to the maximum output size of any supported cipher algorithm.
`output_length`	On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be active, with an IV set if required for the algorithm.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `output` buffer is too small. `PSA_CIPHER_UPDATE_OUTPUT_SIZE()` or `PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE()` can be used to determine the required buffer size.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The following must occur before calling this function:

1. Call either `psa_cipher_encrypt_setup()` or `psa_cipher_decrypt_setup()`. The choice of setup function determines whether this function encrypts or decrypts its input.
2. If the algorithm requires an IV, call `psa_cipher_generate_iv()` or `psa_cipher_set_iv()`. `psa_cipher_generate_iv()` is recommended when encrypting.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_cipher_abort()`.

psa_cipher_finish (function)

Finish encrypting or decrypting a message in a cipher operation.

```
psa_status_t psa_cipher_finish(psa_cipher_operation_t * operation,
                               uint8_t * output,
                               size_t output_size,
                               size_t * output_length);
```

Parameters

operation	Active cipher operation.
output	Buffer where the output is to be written.
output_size	Size of the output buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is PSA_CIPHER_FINISH_OUTPUT_SIZE(key_type, alg) where key_type is the type of key and alg is the algorithm that were used to set up the operation. • PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported cipher algorithm.
output_length	On success, the number of bytes that make up the returned output.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_ARGUMENT	The total input size passed to this operation is not valid for this particular algorithm. For example, the algorithm is a based on block cipher and requires a whole number of blocks, but the total input size is not a multiple of the block size.
PSA_ERROR_INVALID_PADDING	This is a decryption operation for an algorithm that includes padding, and the ciphertext does not contain valid padding.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active, with an IV set if required for the algorithm.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the output buffer is too small. PSA_CIPHER_FINISH_OUTPUT_SIZE() or PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call psa_cipher_encrypt_setup() or psa_cipher_decrypt_setup() before calling this function. The choice of setup function determines whether this function encrypts or decrypts its input.

This function finishes the encryption or decryption of the message formed by concatenating the inputs passed to preceding calls to psa_cipher_update().

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling psa_cipher_abort().

psa_cipher_abort (function)

Abort a cipher operation.

```
psa_status_t psa_cipher_abort(psa_cipher_operation_t * operation);
```

Parameters

operation	Initialized cipher operation.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Aborting an operation frees all associated resources except for the operation object itself. Once aborted, the operation object can be reused for another operation by calling psa_cipher_encrypt_setup() or psa_cipher_decrypt_setup() again.

This function can be called any time after the operation object has been initialized as described in psa_cipher_operation_t.

In particular, calling psa_cipher_abort() after the operation has been terminated by a call to psa_cipher_abort() or psa_cipher_finish() is safe and has no effect.

10.4.4 Support macros

PSA_ALG_IS_STREAM_CIPHER (macro)

Whether the specified algorithm is a stream cipher.

```
#define PSA_ALG_IS_STREAM_CIPHER(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a stream cipher algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier or if it is not a symmetric cipher algorithm.

Description

A stream cipher is a symmetric cipher that encrypts or decrypts messages by applying a bitwise-xor with a stream of bytes that is generated from a key.

PSA_CIPHER_ENCRYPT_OUTPUT_SIZE (macro)

The maximum size of the output of `psa_cipher_encrypt()`, in bytes.

```
#define PSA_CIPHER_ENCRYPT_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
```

Parameters

`key_type`	A symmetric key type that is compatible with algorithm `alg`.
`alg`	A cipher algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_CIPHER(alg)` is true).
`input_length`	Size of the input in bytes.

Returns

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_cipher_encrypt()` will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the output might be smaller.

See also `PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE`.

PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for `psa_cipher_encrypt()`, for any of the supported key types and cipher algorithms.

```
#define PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
```

Parameters

`input_length`	Size of the input in bytes.

Description

If the size of the output buffer is at least this large, it is guaranteed that psa_cipher_encrypt() will not fail due to an insufficient buffer size.

See also PSA_CIPHER_ENCRYPT_OUTPUT_SIZE().

PSA_CIPHER_DECRYPT_OUTPUT_SIZE (macro)

The maximum size of the output of psa_cipher_decrypt(), in bytes.

```
#define PSA_CIPHER_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
```

Parameters

key_type	A symmetric key type that is compatible with algorithm alg.
alg	A cipher algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_CIPHER(alg) is true).
input_length	Size of the input in bytes.

Returns

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

Description

If the size of the output buffer is at least this large, it is guaranteed that psa_cipher_decrypt() will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the output might be smaller.

See also PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE.

PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for psa_cipher_decrypt(), for any of the supported key types and cipher algorithms.

```
#define PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
```

Parameters

input_length	Size of the input in bytes.

Description

If the size of the output buffer is at least this large, it is guaranteed that psa_cipher_decrypt() will not fail due to an insufficient buffer size.

See also PSA_CIPHER_DECRYPT_OUTPUT_SIZE().

PSA_CIPHER_IV_LENGTH (macro)

The default IV size for a cipher algorithm, in bytes.

```
#define PSA_CIPHER_IV_LENGTH(key_type, alg) /* implementation-defined value */
```

Parameters

`key_type`	A symmetric key type that is compatible with algorithm `alg`.
`alg`	A cipher algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_CIPHER(alg)` is true).

Returns

The default IV size for the specified key type and algorithm. If the algorithm does not use an IV, return `0`. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

Description

The IV that is generated as part of a call to `psa_cipher_encrypt()` is always the default IV length for the algorithm.

This macro can be used to allocate a buffer of sufficient size to store the IV output from `psa_cipher_generate_iv()` when using a multi-part cipher operation.

See also `PSA_CIPHER_IV_MAX_SIZE`.

PSA_CIPHER_IV_MAX_SIZE (macro)

The maximum IV size for all supported cipher algorithms, in bytes.

```
#define PSA_CIPHER_IV_MAX_SIZE /* implementation-defined value */
```

See also `PSA_CIPHER_IV_LENGTH()`.

PSA_CIPHER_UPDATE_OUTPUT_SIZE (macro)

A sufficient output buffer size for `psa_cipher_update()`.

```
#define PSA_CIPHER_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
```

Parameters

`key_type`	A symmetric key type that is compatible with algorithm `alg`.
`alg`	A cipher algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_CIPHER(alg)` is true).
`input_length`	Size of the input in bytes.

Returns

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_cipher_update()` will not fail due to an insufficient buffer size. The actual size of the output might be smaller in any given call.

See also `PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE`.

`PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE` (macro)

A sufficient output buffer size for `psa_cipher_update()`, for any of the supported key types and cipher algorithms.

```
#define PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
```

Parameters

`input_length`	Size of the input in bytes.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_cipher_update()` will not fail due to an insufficient buffer size.

See also `PSA_CIPHER_UPDATE_OUTPUT_SIZE()`.

`PSA_CIPHER_FINISH_OUTPUT_SIZE` (macro)

A sufficient ciphertext buffer size for `psa_cipher_finish()`.

```
#define PSA_CIPHER_FINISH_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
```

Parameters

`key_type`	A symmetric key type that is compatible with algorithm `alg`.
`alg`	A cipher algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_CIPHER(alg)` is true).

Returns

A sufficient output size for the specified key type and algorithm. If the key type or cipher algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and cipher algorithm that it recognizes, but does not support.

Description

If the size of the ciphertext buffer is at least this large, it is guaranteed that `psa_cipher_finish()` will not fail due to an insufficient ciphertext buffer size. The actual size of the output might be smaller in any given call.

See also `PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE`.

PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE (macro)

A sufficient ciphertext buffer size for psa_cipher_finish(), for any of the supported key types and cipher algorithms.

```
#define PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE /* implementation-defined value */
```

See also PSA_CIPHER_FINISH_OUTPUT_SIZE().

PSA_BLOCK_CIPHER_BLOCK_LENGTH (macro)

The block size of a block cipher.

```
#define PSA_BLOCK_CIPHER_BLOCK_LENGTH(type) /* specification-defined value */
```

Parameters

type	A cipher key type (value of type psa_key_type_t).

Returns

The block size for a block cipher, or 1 for a stream cipher. The return value is undefined if type is not a supported cipher key type.

Description

> **Note:**
>
> It is possible to build stream cipher algorithms on top of a block cipher, for example CTR mode (PSA_ALG_CTR). This macro only takes the key type into account, so it cannot be used to determine the size of the data that psa_cipher_update() might buffer for future processing in general.

> **Note:**
>
> This macro expression is a compile-time constant if type is a compile-time constant.

> **Warning:** This macro is permitted to evaluate its argument multiple times.

See also PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE.

PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE (macro)

The maximum size of a block cipher supported by the implementation.

```
#define PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE /* implementation-defined value */
```

See also PSA_BLOCK_CIPHER_BLOCK_LENGTH().

10.5 Authenticated encryption with associated data (AEAD)

10.5.1 AEAD algorithms

PSA_ALG_CCM (macro)

The *Counter with CBC-MAC* (CCM) authenticated encryption algorithm.

```
#define PSA_ALG_CCM ((psa_algorithm_t)0x05500100)
```

CCM is defined for block ciphers that have a 128-bit block size. The underlying block cipher is determined by the key type.

To use PSA_ALG_CCM with a multi-part AEAD operation, the application must call psa_aead_set_lengths() before providing the nonce, the additional data and plaintext to the operation.

CCM requires a nonce of between 7 and 13 bytes in length. The length of the nonce depends on the length of the plaintext:

- CCM encodes the plaintext length *pLen* in *L* octets, with *L* the smallest integer >= 2 where *pLen* < 2^(8*L*).
- The nonce length is then 15 - *L* bytes.

If the application is generating a random nonce using psa_aead_generate_nonce(), the size of the generated nonce is 15 - *L* bytes.

CCM supports authentication tag sizes of 4, 6, 8, 10, 12, 14, and 16 bytes. The default tag length is 16. Shortened tag lengths can be requested using PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CCM, tag_length), where tag_length is a valid CCM tag length.

The CCM block cipher mode is defined in *Counter with CBC-MAC (CCM)* [RFC3610].

PSA_ALG_GCM (macro)

The *Galois/Counter Mode* (GCM) authenticated encryption algorithm.

```
#define PSA_ALG_GCM ((psa_algorithm_t)0x05500200)
```

GCM is defined for block ciphers that have a 128-bit block size. The underlying block cipher is determined by the key type.

GCM requires a nonce of at least 1 byte in length. The maximum supported nonce size is IMPLEMENTATION DEFINED. Calling psa_aead_generate_nonce() will generate a random 12-byte nonce.

GCM supports authentication tag sizes of 4, 8, 12, 13, 14, 15, and 16 bytes. The default tag length is 16. Shortened tag lengths can be requested using PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_GCM, tag_length), where tag_length is a valid GCM tag length.

The GCM block cipher mode is defined in *NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC* [SP800-38D].

PSA_ALG_CHACHA20_POLY1305 (macro)

The ChaCha20-Poly1305 AEAD algorithm.

```
#define PSA_ALG_CHACHA20_POLY1305 ((psa_algorithm_t)0x05100500)
```

There are two defined variants of ChaCha20-Poly1305:

- An implementation that supports ChaCha20-Poly1305 must support the variant defined by *ChaCha20 and Poly1305 for IETF Protocols* [RFC7539], which has a 96-bit nonce and 32-bit counter.
- An implementation can optionally also support the original variant defined by *ChaCha, a variant of Salsa20* [CHACHA20], which has a 64-bit nonce and 64-bit counter.

The variant used for the AEAD encryption or decryption operation, depends on the nonce provided for an AEAD operation using `PSA_ALG_CHACHA20_POLY1305`:

- A nonce provided in a call to `psa_aead_encrypt()`, `psa_aead_decrypt()` or `psa_aead_set_nonce()` must be 8 or 12 bytes. The size of nonce will select the appropriate variant of the algorithm.
- A nonce generated by a call to `psa_aead_generate_nonce()` will be 12 bytes, and will use the [RFC7539] variant.

Implementations must support 16-byte tags. It is recommended that truncated tag sizes are rejected.

`PSA_ALG_AEAD_WITH_SHORTENED_TAG` (macro)

Macro to build a AEAD algorithm with a shortened tag.

```
#define PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, tag_length) \
    /* specification-defined value */
```

Parameters

`aead_alg`	An AEAD algorithm identifier (value of type `psa_algorithm_t` such that `PSA_ALG_IS_AEAD(alg)` is true).
`tag_length`	Desired length of the authentication tag in bytes.

Returns

The corresponding AEAD algorithm with the specified tag length.

Unspecified if `alg` is not a supported AEAD algorithm or if `tag_length` is not valid for the specified AEAD algorithm.

Description

An AEAD algorithm with a shortened tag is similar to the corresponding AEAD algorithm, but has an authentication tag that consists of fewer bytes. Depending on the algorithm, the tag length might affect the calculation of the ciphertext.

The AEAD algorithm with a default length tag can be recovered using `PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG()`.

10.5.2 Single-part AEAD functions

`psa_aead_encrypt` (function)

Process an authenticated encryption operation.

```
psa_status_t psa_aead_encrypt(psa_key_id_t key,
                              psa_algorithm_t alg,
                              const uint8_t * nonce,
                              size_t nonce_length,
                              const uint8_t * additional_data,
                              size_t additional_data_length,
                              const uint8_t * plaintext,
                              size_t plaintext_length,
                              uint8_t * ciphertext,
                              size_t ciphertext_size,
                              size_t * ciphertext_length);
```

Parameters

`key`	Identifier of the key to use for the operation. It must allow the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_AEAD(alg)` is true).
`nonce`	Nonce or IV to use.
`nonce_length`	Size of the `nonce` buffer in bytes. This must be appropriate for the selected algorithm. The default nonce size is `PSA_AEAD_NONCE_LENGTH(key_type, alg)` where `key_type` is the type of `key`.
`additional_data`	Additional data that will be authenticated but not encrypted.
`additional_data_length`	Size of `additional_data` in bytes.
`plaintext`	Data that will be authenticated and encrypted.
`plaintext_length`	Size of `plaintext` in bytes.
`ciphertext`	Output buffer for the authenticated and encrypted data. The additional data is not part of this output. For algorithms where the encrypted data and the authentication tag are defined as separate outputs, the authentication tag is appended to the encrypted data.
`ciphertext_size`	Size of the `ciphertext` buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is `PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length)` where `key_type` is the type of `key`. • `PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length)` evaluates to the maximum ciphertext size of any supported AEAD encryption.
`ciphertext_length`	On success, the size of the output in the `ciphertext` buffer.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_ENCRYPT` flag, or it does not

	permit the requested algorithm.
PSA_ERROR_INVALID_ARGUMENT	key is not compatible with alg.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not an AEAD algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_BUFFER_TOO_SMALL	ciphertext_size is too small. PSA_AEAD_ENCRYPT_OUTPUT_SIZE() or PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE() can be used to determine the required buffer size.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

psa_aead_decrypt (function)

Process an authenticated decryption operation.

```
psa_status_t psa_aead_decrypt(psa_key_id_t key,
                              psa_algorithm_t alg,
                              const uint8_t * nonce,
                              size_t nonce_length,
                              const uint8_t * additional_data,
                              size_t additional_data_length,
                              const uint8_t * ciphertext,
                              size_t ciphertext_length,
                              uint8_t * plaintext,
                              size_t plaintext_size,
                              size_t * plaintext_length);
```

Parameters

key	Identifier of the key to use for the operation. It must allow the usage PSA_KEY_USAGE_DECRYPT.
alg	The AEAD algorithm to compute (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
nonce	Nonce or IV to use.
nonce_length	Size of the nonce buffer in bytes. This must be appropriate for the selected algorithm. The default nonce size is PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type is the type of key.
additional_data	Additional data that has been authenticated but not encrypted.

additional_data_length	Size of additional_data in bytes.
ciphertext	Data that has been authenticated and encrypted. For algorithms where the encrypted data and the authentication tag are defined as separate inputs, the buffer must contain the encrypted data followed by the authentication tag.
ciphertext_length	Size of ciphertext in bytes.
plaintext	Output buffer for the decrypted data.
plaintext_size	Size of the plaintext buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg, ciphertext_length) where key_type is the type of key. • PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length) evaluates to the maximum plaintext size of any supported AEAD decryption.
plaintext_length	On success, the size of the output in the plaintext buffer.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_INVALID_SIGNATURE	The ciphertext is not authentic.
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_DECRYPT flag, or it does not permit the requested algorithm.
PSA_ERROR_INVALID_ARGUMENT	key is not compatible with alg.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not an AEAD algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_BUFFER_TOO_SMALL	plaintext_size is too small. PSA_AEAD_DECRYPT_OUTPUT_SIZE() or PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE() can be used to determine the required buffer size.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

10.5.3 Multi-part AEAD operations

Warning: When decrypting using a multi-part AEAD operation, there is no guarantee that the input or output is valid until `psa_aead_verify()` has returned `PSA_SUCCESS`.

A call to `psa_aead_update()` or `psa_aead_update_ad()` returning `PSA_SUCCESS` **does not** indicate that the input and output is valid.

Until an application calls `psa_aead_verify()` and it has returned `PSA_SUCCESS`, the following rules apply to input and output data from a multi-part AEAD operation:

- Do not trust the input. If the application takes any action that depends on the input data, this action will need to be undone if the input turns out to be invalid.
- Store the output in a confidential location. In particular, the application must not copy the output to a memory or storage space which is shared.
- Do not trust the output. If the application takes any action that depends on the tentative decrypted data, this action will need to be undone if the input turns out to be invalid. Furthermore, if an adversary can observe that this action took place, for example, through timing, they might be able to use this fact as an oracle to decrypt any message encrypted with the same key.

An application that does not follow these rules might be vulnerable to maliciously constructed AEAD input data.

`psa_aead_operation_t` (type)

The type of the state object for multi-part AEAD operations.

```
typedef /* implementation-defined type */ psa_aead_operation_t;
```

Before calling any function on an AEAD operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_aead_operation_t operation;
  memset(&operation, 0, sizeof(operation));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

  ```
  static psa_aead_operation_t operation;
  ```

- Initialize the object to the initializer `PSA_AEAD_OPERATION_INIT`, for example:

  ```
  psa_aead_operation_t operation = PSA_AEAD_OPERATION_INIT;
  ```

- Assign the result of the function `psa_aead_operation_init()` to the object, for example:

```
psa_aead_operation_t operation;
operation = psa_aead_operation_init();
```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

PSA_AEAD_OPERATION_INIT (macro)

This macro returns a suitable initializer for an AEAD operation object of type `psa_aead_operation_t`.

```
#define PSA_AEAD_OPERATION_INIT /* implementation-defined value */
```

psa_aead_operation_init (function)

Return an initial value for an AEAD operation object.

```
psa_aead_operation_t psa_aead_operation_init(void);
```

Returns: `psa_aead_operation_t`

psa_aead_encrypt_setup (function)

Set the key for a multi-part authenticated encryption operation.

```
psa_status_t psa_aead_encrypt_setup(psa_aead_operation_t * operation,
                                    psa_key_id_t key,
                                    psa_algorithm_t alg);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_aead_operation_t` and not yet in use.
`key`	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_AEAD(alg)` is true).

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be inactive.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_ENCRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not an AEAD algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	

PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The sequence of operations to encrypt a message with authentication is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for psa_aead_operation_t, e.g. PSA_AEAD_OPERATION_INIT.
3. Call psa_aead_encrypt_setup() to specify the algorithm and key.
4. If needed, call psa_aead_set_lengths() to specify the length of the inputs to the subsequent calls to psa_aead_update_ad() and psa_aead_update(). See the documentation of psa_aead_set_lengths() for details.
5. Call either psa_aead_generate_nonce() or psa_aead_set_nonce() to generate or set the nonce. It is recommended to use psa_aead_generate_nonce() unless the protocol being implemented requires a specific nonce value.
6. Call psa_aead_update_ad() zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
7. Call psa_aead_update() zero, one or more times, passing a fragment of the message to encrypt each time.
8. Call psa_aead_finish().

If an error occurs at any step after a call to psa_aead_encrypt_setup(), the operation will need to be reset by a call to psa_aead_abort(). The application can call psa_aead_abort() at any time after the operation has been initialized.

After a successful call to psa_aead_encrypt_setup(), the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to psa_aead_finish().
- A call to psa_aead_abort().

psa_aead_decrypt_setup (function)

Set the key for a multi-part authenticated decryption operation.

```
psa_status_t psa_aead_decrypt_setup(psa_aead_operation_t * operation,
                                    psa_key_id_t key,
                                    psa_algorithm_t alg);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_aead_operation_t` and not yet in use.
`key`	Identifier of the key to use for the operation. It must remain valid until the operation terminates. It must allow the usage `PSA_KEY_USAGE_DECRYPT`.
`alg`	The AEAD algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_AEAD(alg)` is true).

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be inactive.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_DECRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not an AEAD algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The sequence of operations to decrypt a message with authentication is as follows:

1. Allocate an operation object which will be passed to all the functions listed here.
2. Initialize the operation object with one of the methods described in the documentation for `psa_aead_operation_t`, e.g. `PSA_AEAD_OPERATION_INIT`.
3. Call `psa_aead_decrypt_setup()` to specify the algorithm and key.
4. If needed, call `psa_aead_set_lengths()` to specify the length of the inputs to the subsequent calls to `psa_aead_update_ad()` and `psa_aead_update()`. See the documentation of `psa_aead_set_lengths()` for details.

5. Call psa_aead_set_nonce() with the nonce for the decryption.
6. Call psa_aead_update_ad() zero, one or more times, passing a fragment of the non-encrypted additional authenticated data each time.
7. Call psa_aead_update() zero, one or more times, passing a fragment of the ciphertext to decrypt each time.
8. Call psa_aead_verify().

If an error occurs at any step after a call to psa_aead_decrypt_setup(), the operation will need to be reset by a call to psa_aead_abort(). The application can call psa_aead_abort() at any time after the operation has been initialized.

After a successful call to psa_aead_decrypt_setup(), the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to psa_aead_verify().
- A call to psa_aead_abort().

psa_aead_set_lengths (function)

Declare the lengths of the message and additional data for AEAD.

```
psa_status_t psa_aead_set_lengths(psa_aead_operation_t * operation,
                                  size_t ad_length,
                                  size_t plaintext_length);
```

Parameters

operation	Active AEAD operation.
ad_length	Size of the non-encrypted additional authenticated data in bytes.
plaintext_length	Size of the plaintext to encrypt in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active, and psa_aead_set_nonce() and psa_aead_generate_nonce() must not have been called yet.
PSA_ERROR_INVALID_ARGUMENT	At least one of the lengths is not acceptable for the chosen algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The application must call this function before calling psa_aead_set_nonce() or psa_aead_generate_nonce(), if the algorithm for the operation requires it. If the algorithm does not require it, calling this function is optional, but if this function is called then the implementation must enforce the lengths.

- For PSA_ALG_CCM, calling this function is required.
- For the other AEAD algorithms defined in this specification, calling this function is not required.
- For vendor-defined algorithm, refer to the vendor documentation.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

psa_aead_generate_nonce (function)

Generate a random nonce for an authenticated encryption operation.

```
psa_status_t psa_aead_generate_nonce(psa_aead_operation_t * operation,
                                     uint8_t * nonce,
                                     size_t nonce_size,
                                     size_t * nonce_length);
```

Parameters

operation	Active AEAD operation.
nonce	Buffer where the generated nonce is to be written.
nonce_size	Size of the nonce buffer in bytes. This must be at least PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type and alg are type of key and the algorithm respectively that were used to set up the AEAD operation.
nonce_length	On success, the number of bytes of the generated nonce.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be an active AEAD encryption operation, with no nonce set.
PSA_ERROR_BAD_STATE	The operation state is not valid: this is an algorithm which requires psa_aead_set_lengths() to be called before setting the nonce.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the nonce buffer is too small. PSA_AEAD_NONCE_LENGTH() or PSA_AEAD_NONCE_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	

PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function generates a random nonce for the authenticated encryption operation with an appropriate size for the chosen algorithm, key type and key size.

The application must call psa_aead_encrypt_setup() before calling this function. If applicable for the algorithm, the application must call psa_aead_set_lengths() before calling this function.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

psa_aead_set_nonce (function)

Set the nonce for an authenticated encryption or decryption operation.

```
psa_status_t psa_aead_set_nonce(psa_aead_operation_t * operation,
                                const uint8_t * nonce,
                                size_t nonce_length);
```

Parameters

operation	Active AEAD operation.
nonce	Buffer containing the nonce to use.
nonce_length	Size of the nonce in bytes. This must be a valid nonce size for the chosen algorithm. The default nonce size is PSA_AEAD_NONCE_LENGTH(key_type, alg) where key_type and alg are type of key and the algorithm respectively that were used to set up the AEAD operation.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active, with no nonce set.
PSA_ERROR_BAD_STATE	The operation state is not valid: this is an algorithm which requires psa_aead_set_lengths() to be called before setting the nonce.
PSA_ERROR_INVALID_ARGUMENT	The size of nonce is not acceptable for the chosen algorithm.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	

PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function sets the nonce for the authenticated encryption or decryption operation.

The application must call psa_aead_encrypt_setup() or psa_aead_decrypt_setup() before calling this function. If applicable for the algorithm, the application must call psa_aead_set_lengths() before calling this function.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

> **Note:**
>
> When encrypting, psa_aead_generate_nonce() is recommended instead of using this function, unless implementing a protocol that requires a non-random IV.

psa_aead_update_ad (function)

Pass additional data to an active AEAD operation.

```
psa_status_t psa_aead_update_ad(psa_aead_operation_t * operation,
                                const uint8_t * input,
                                size_t input_length);
```

Parameters

operation	Active AEAD operation.
input	Buffer containing the fragment of additional data.
input_length	Size of the input buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success. **Warning:** When decrypting, do not trust the input until psa_aead_verify() succeeds. See the detailed warning.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active, have a nonce set, have lengths set if required by the algorithm, and psa_aead_update() must not have been called yet.
PSA_ERROR_INVALID_ARGUMENT	The total input length overflows the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	

PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Additional data is authenticated, but not encrypted.

This function can be called multiple times to pass successive fragments of the additional data. This function must not be called after passing data to encrypt or decrypt with psa_aead_update().

The following must occur before calling this function:

1. Call either psa_aead_encrypt_setup() or psa_aead_decrypt_setup().
2. Set the nonce with psa_aead_generate_nonce() or psa_aead_set_nonce().

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

psa_aead_update (function)

Encrypt or decrypt a message fragment in an active AEAD operation.

```
psa_status_t psa_aead_update(psa_aead_operation_t * operation,
                             const uint8_t * input,
                             size_t input_length,
                             uint8_t * output,
                             size_t output_size,
                             size_t * output_length);
```

Parameters

operation	Active AEAD operation.
input	Buffer containing the message fragment to encrypt or decrypt.
input_length	Size of the input buffer in bytes.
output	Buffer where the output is to be written.
output_size	Size of the output buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is PSA_AEAD_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) where key_type is the type of key and alg is the algorithm that were used to set up the operation. • PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported AEAD algorithm.

output_length	On success, the number of bytes that make up the returned output.

Returns: psa_status_t

PSA_SUCCESS	Success. **Warning:** When decrypting, do not use the output until psa_aead_verify() succeeds. See the detailed warning.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active, have a nonce set, and have lengths set if required by the algorithm.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the output buffer is too small. PSA_AEAD_UPDATE_OUTPUT_SIZE() or PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE() can be used to determine the required buffer size.
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update_ad() so far is less than the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INVALID_ARGUMENT	The total input length overflows the plaintext length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The following must occur before calling this function:

1. Call either psa_aead_encrypt_setup() or psa_aead_decrypt_setup(). The choice of setup function determines whether this function encrypts or decrypts its input.
2. Set the nonce with psa_aead_generate_nonce() or psa_aead_set_nonce().
3. Call psa_aead_update_ad() to pass all the additional data.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

This function does not require the input to be aligned to any particular block boundary. If the implementation can only process a whole block at a time, it must consume all the input provided, but it might delay the end of the corresponding output until a subsequent call to psa_aead_update(), psa_aead_finish() or psa_aead_verify() provides sufficient input. The amount of data that can be delayed in this way is bounded by PSA_AEAD_UPDATE_OUTPUT_SIZE().

psa_aead_finish (function)

Finish encrypting a message in an AEAD operation.

```
psa_status_t psa_aead_finish(psa_aead_operation_t * operation,
                             uint8_t * ciphertext,
                             size_t ciphertext_size,
                             size_t * ciphertext_length,
                             uint8_t * tag,
                             size_t tag_size,
                             size_t * tag_length);
```

Parameters

operation	Active AEAD operation.
ciphertext	Buffer where the last part of the ciphertext is to be written.
ciphertext_size	Size of the ciphertext buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is PSA_AEAD_FINISH_OUTPUT_SIZE(key_type, alg) where key_type is the type of key and alg is the algorithm that were used to set up the operation. • PSA_AEAD_FINISH_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported AEAD algorithm.
ciphertext_length	On success, the number of bytes of returned ciphertext.
tag	Buffer where the authentication tag is to be written.
tag_size	Size of the tag buffer in bytes. This must be appropriate for the selected algorithm and key: • The exact tag size is PSA_AEAD_TAG_LENGTH(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size of the key, and alg is the algorithm that were used in the call to psa_aead_encrypt_setup(). • PSA_AEAD_TAG_MAX_SIZE evaluates to the maximum tag size of any supported AEAD algorithm.
tag_length	On success, the number of bytes that make up the returned tag.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be an active encryption operation with a nonce set.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the ciphertext or tag buffer is too small. PSA_AEAD_FINISH_OUTPUT_SIZE() or PSA_AEAD_FINISH_OUTPUT_MAX_SIZE can be used to determine the required ciphertext buffer size.

	PSA_AEAD_TAG_LENGTH() or PSA_AEAD_TAG_MAX_SIZE can be used to determine the required tag buffer size.
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update_ad() so far is less than the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update() so far is less than the plaintext length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The operation must have been set up with psa_aead_encrypt_setup().

This function finishes the authentication of the additional data formed by concatenating the inputs passed to preceding calls to psa_aead_update_ad() with the plaintext formed by concatenating the inputs passed to preceding calls to psa_aead_update().

This function has two output buffers:

- ciphertext contains trailing ciphertext that was buffered from preceding calls to psa_aead_update().
- tag contains the authentication tag.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

psa_aead_verify (function)

Finish authenticating and decrypting a message in an AEAD operation.

```
psa_status_t psa_aead_verify(psa_aead_operation_t * operation,
                             uint8_t * plaintext,
                             size_t plaintext_size,
                             size_t * plaintext_length,
                             const uint8_t * tag,
                             size_t tag_length);
```

Parameters

operation	Active AEAD operation.

plaintext	Buffer where the last part of the plaintext is to be written. This is the remaining data from previous calls to psa_aead_update() that could not be processed until the end of the input.
plaintext_size	Size of the plaintext buffer in bytes. This must be appropriate for the selected algorithm and key: • A sufficient output size is PSA_AEAD_VERIFY_OUTPUT_SIZE(key_type, alg) where key_type is the type of key and alg is the algorithm that were used to set up the operation. • PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported AEAD algorithm.
plaintext_length	On success, the number of bytes of returned plaintext.
tag	Buffer containing the authentication tag.
tag_length	Size of the tag buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_SIGNATURE	The calculations were successful, but the authentication tag is not correct.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be an active decryption operation with a nonce set.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the plaintext buffer is too small. PSA_AEAD_VERIFY_OUTPUT_SIZE() or PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update_ad() so far is less than the additional data length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INVALID_ARGUMENT	The total length of input to psa_aead_update() so far is less than the plaintext length that was previously specified with psa_aead_set_lengths().
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The operation must have been set up with psa_aead_decrypt_setup().

This function finishes the authenticated decryption of the message components:

- The additional data consisting of the concatenation of the inputs passed to preceding calls to psa_aead_update_ad().
- The ciphertext consisting of the concatenation of the inputs passed to preceding calls to psa_aead_update().
- The tag passed to this function call.

If the authentication tag is correct, this function outputs any remaining plaintext and reports success. If the authentication tag is not correct, this function returns PSA_ERROR_INVALID_SIGNATURE.

When this function returns successfully, the operation becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling psa_aead_abort().

> **Note:**
>
> Implementations must make the best effort to ensure that the comparison between the actual tag and the expected tag is performed in constant time.

psa_aead_abort (function)

Abort an AEAD operation.

```
psa_status_t psa_aead_abort(psa_aead_operation_t * operation);
```

Parameters

operation	Initialized AEAD operation.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Aborting an operation frees all associated resources except for the operation object itself. Once aborted, the operation object can be reused for another operation by calling psa_aead_encrypt_setup() or psa_aead_decrypt_setup() again.

This function can be called any time after the operation object has been initialized as described in psa_aead_operation_t.

In particular, calling psa_aead_abort() after the operation has been terminated by a call to psa_aead_abort(), psa_aead_finish() or psa_aead_verify() is safe and has no effect.

10.5.4 Support macros

PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER (macro)

Whether the specified algorithm is an AEAD mode on a block cipher.

```
#define PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is an AEAD algorithm which is an AEAD mode based on a block cipher, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG (macro)

An AEAD algorithm with the default tag length.

```
#define PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG(aead_alg) \
    /* specification-defined value */
```

Parameters

aead_alg	An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

Returns

The corresponding AEAD algorithm with the default tag length for that algorithm.

Description

This macro can be used to construct the AEAD algorithm with default tag length from an AEAD algorithm with a shortened tag. See also PSA_ALG_AEAD_WITH_SHORTENED_TAG().

PSA_AEAD_ENCRYPT_OUTPUT_SIZE (macro)

The maximum size of the output of psa_aead_encrypt(), in bytes.

```
#define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length) \
    /* implementation-defined value */
```

Parameters

key_type	A symmetric key type that is compatible with algorithm alg.
alg	An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
plaintext_length	Size of the plaintext in bytes.

Returns

The AEAD ciphertext size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

Description

If the size of the ciphertext buffer is at least this large, it is guaranteed that `psa_aead_encrypt()` will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the ciphertext might be smaller.

See also `PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE`.

PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for `psa_aead_encrypt()`, for any of the supported key types and AEAD algorithms.

```
#define PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length) \
    /* implementation-defined value */
```

Parameters

`plaintext_length`	Size of the plaintext in bytes.

Description

If the size of the ciphertext buffer is at least this large, it is guaranteed that `psa_aead_encrypt()` will not fail due to an insufficient buffer size.

See also `PSA_AEAD_ENCRYPT_OUTPUT_SIZE()`.

PSA_AEAD_DECRYPT_OUTPUT_SIZE (macro)

The maximum size of the output of `psa_aead_decrypt()`, in bytes.

```
#define PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg, ciphertext_length) \
    /* implementation-defined value */
```

Parameters

`key_type`	A symmetric key type that is compatible with algorithm `alg`.
`alg`	An AEAD algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_AEAD(alg)` is true).
`ciphertext_length`	Size of the ciphertext in bytes.

Returns

The AEAD plaintext size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

Description

If the size of the plaintext buffer is at least this large, it is guaranteed that `psa_aead_decrypt()` will not fail due to an insufficient buffer size. Depending on the algorithm, the actual size of the plaintext might be smaller.

See also `PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE`.

PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for `psa_aead_decrypt()`, for any of the supported key types and AEAD algorithms.

```
#define PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length) \
    /* implementation-defined value */
```

Parameters

`ciphertext_length`	Size of the ciphertext in bytes.

Description

If the size of the plaintext buffer is at least this large, it is guaranteed that `psa_aead_decrypt()` will not fail due to an insufficient buffer size.

See also `PSA_AEAD_DECRYPT_OUTPUT_SIZE()`.

PSA_AEAD_NONCE_LENGTH (macro)

The default nonce size for an AEAD algorithm, in bytes.

```
#define PSA_AEAD_NONCE_LENGTH(key_type, alg) /* implementation-defined value */
```

Parameters

`key_type`	A symmetric key type that is compatible with algorithm `alg`.
`alg`	An AEAD algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_AEAD(alg)` is true).

Returns

The default nonce size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return `0`. An implementation can return either `0` or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

Description

This macro can be used to allocate a buffer of sufficient size to store the nonce output from `psa_aead_generate_nonce()`.

See also `PSA_AEAD_NONCE_MAX_SIZE`.

PSA_AEAD_NONCE_MAX_SIZE (macro)

The maximum nonce size for all supported AEAD algorithms, in bytes.

```
#define PSA_AEAD_NONCE_MAX_SIZE /* implementation-defined value */
```

See also PSA_AEAD_NONCE_LENGTH().

PSA_AEAD_UPDATE_OUTPUT_SIZE (macro)

A sufficient output buffer size for psa_aead_update().

```
#define PSA_AEAD_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
```

Parameters

key_type	A symmetric key type that is compatible with algorithm alg.
alg	An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).
input_length	Size of the input in bytes.

Returns

A sufficient output buffer size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

Description

If the size of the output buffer is at least this large, it is guaranteed that psa_aead_update() will not fail due to an insufficient buffer size. The actual size of the output might be smaller in any given call.

See also PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE.

PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for psa_aead_update(), for any of the supported key types and AEAD algorithms.

```
#define PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
```

Parameters

input_length	Size of the input in bytes.

Description

If the size of the output buffer is at least this large, it is guaranteed that psa_aead_update() will not fail due to an insufficient buffer size.

See also PSA_AEAD_UPDATE_OUTPUT_SIZE().

PSA_AEAD_FINISH_OUTPUT_SIZE (macro)

A sufficient ciphertext buffer size for psa_aead_finish().

```
#define PSA_AEAD_FINISH_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
```

Parameters

key_type	A symmetric key type that is compatible with algorithm alg.
alg	An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

Returns

A sufficient ciphertext buffer size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

Description

If the size of the ciphertext buffer is at least this large, it is guaranteed that psa_aead_finish() will not fail due to an insufficient ciphertext buffer size. The actual size of the output might be smaller in any given call.

See also PSA_AEAD_FINISH_OUTPUT_MAX_SIZE.

PSA_AEAD_FINISH_OUTPUT_MAX_SIZE (macro)

A sufficient ciphertext buffer size for psa_aead_finish(), for any of the supported key types and AEAD algorithms.

```
#define PSA_AEAD_FINISH_OUTPUT_MAX_SIZE /* implementation-defined value */
```

See also PSA_AEAD_FINISH_OUTPUT_SIZE().

PSA_AEAD_TAG_LENGTH (macro)

The length of a tag for an AEAD algorithm, in bytes.

```
#define PSA_AEAD_TAG_LENGTH(key_type, key_bits, alg) \
    /* implementation-defined value */
```

Parameters

key_type	The type of the AEAD key.
key_bits	The size of the AEAD key in bits.
alg	An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

Returns

The tag length for the specified algorithm and key. If the AEAD algorithm does not have an identified tag that can be distinguished from the rest of the ciphertext, return 0. If the AEAD algorithm is not recognized, return 0. An implementation can return either 0 or a correct size for an AEAD algorithm that it recognizes, but does not support.

Description

This macro can be used to allocate a buffer of sufficient size to store the tag output from psa_aead_finish().

See also PSA_AEAD_TAG_MAX_SIZE.

PSA_AEAD_TAG_MAX_SIZE (macro)

The maximum tag size for all supported AEAD algorithms, in bytes.

```
#define PSA_AEAD_TAG_MAX_SIZE /* implementation-defined value */
```

See also PSA_AEAD_TAG_LENGTH().

PSA_AEAD_VERIFY_OUTPUT_SIZE (macro)

A sufficient plaintext buffer size for psa_aead_verify().

```
#define PSA_AEAD_VERIFY_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
```

Parameters

key_type	A symmetric key type that is compatible with algorithm alg.
alg	An AEAD algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_AEAD(alg) is true).

Returns

A sufficient plaintext buffer size for the specified key type and algorithm. If the key type or AEAD algorithm is not recognized, or the parameters are incompatible, return 0. An implementation can return either 0 or a correct size for a key type and AEAD algorithm that it recognizes, but does not support.

Description

If the size of the plaintext buffer is at least this large, it is guaranteed that psa_aead_verify() will not fail due to an insufficient plaintext buffer size. The actual size of the output might be smaller in any given call.

See also PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE.

PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE (macro)

A sufficient plaintext buffer size for psa_aead_verify(), for any of the supported key types and AEAD algorithms.

```
#define PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE /* implementation-defined value */
```

See also PSA_AEAD_VERIFY_OUTPUT_SIZE().

10.6 Key derivation

10.6.1 Key derivation algorithms

PSA_ALG_HKDF (macro)

Macro to build an HKDF algorithm.

```
#define PSA_ALG_HKDF(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true).

Returns

The corresponding HKDF algorithm. For example, `PSA_ALG_HKDF(PSA_ALG_SHA_256)` is HKDF using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is the HMAC-based Extract-and-Expand Key Derivation Function (HKDF) specified by *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [RFC5869].

This key derivation algorithm uses the following inputs:

- `PSA_KEY_DERIVATION_INPUT_SALT` is the salt used in the "extract" step. It is optional; if omitted, the derivation uses an empty salt.
- `PSA_KEY_DERIVATION_INPUT_SECRET` is the secret key used in the "extract" step.
- `PSA_KEY_DERIVATION_INPUT_INFO` is the info string used in the "expand" step.

If `PSA_KEY_DERIVATION_INPUT_SALT` is provided, it must be before `PSA_KEY_DERIVATION_INPUT_SECRET`. `PSA_KEY_DERIVATION_INPUT_INFO` can be provided at any time after setup and before starting to generate output.

Each input may only be passed once.

PSA_ALG_TLS12_PRF (macro)

Macro to build a TLS-1.2 PRF algorithm.

```
#define PSA_ALG_TLS12_PRF(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true).

Returns

The corresponding TLS-1.2 PRF algorithm. For example, `PSA_ALG_TLS12_PRF(PSA_ALG_SHA_256)` represents the TLS 1.2 PRF using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

TLS 1.2 uses a custom pseudorandom function (PRF) for key schedule, specified in *The Transport Layer Security (TLS) Protocol Version 1.2* [RFC5246] §5. It is based on HMAC and can be used with either SHA-256 or SHA-384.

This key derivation algorithm uses the following inputs, which must be passed in the order given here:

- PSA_KEY_DERIVATION_INPUT_SEED is the seed.
- PSA_KEY_DERIVATION_INPUT_SECRET is the secret key.
- PSA_KEY_DERIVATION_INPUT_LABEL is the label.

Each input may only be passed once.

For the application to TLS-1.2 key expansion:

- The seed is the concatenation of `ServerHello.Random + ClientHello.Random`.
- The label is `"key expansion"`.

PSA_ALG_TLS12_PSK_TO_MS (macro)

Macro to build a TLS-1.2 PSK-to-MasterSecret algorithm.

```
#define PSA_ALG_TLS12_PSK_TO_MS(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true).

Returns

The corresponding TLS-1.2 PSK to MS algorithm. For example, `PSA_ALG_TLS12_PSK_TO_MS(PSA_ALG_SHA_256)` represents the TLS-1.2 PSK to MasterSecret derivation PRF using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

In a pure-PSK handshake in TLS 1.2, the master secret (MS) is derived from the pre-shared key (PSK) through the application of padding (*Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)* [RFC4279] §2) and the TLS-1.2 PRF (*The Transport Layer Security (TLS) Protocol Version 1.2* [RFC5246] §5). The latter is based on HMAC and can be used with either SHA-256 or SHA-384.

This key derivation algorithm uses the following inputs, which must be passed in the order given here:

- PSA_KEY_DERIVATION_INPUT_SEED is the seed.
- PSA_KEY_DERIVATION_INPUT_SECRET is the PSK. The PSK must not be larger than PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE.
- PSA_KEY_DERIVATION_INPUT_LABEL is the label.

Each input may only be passed once.

For the application to TLS-1.2:

- The seed, which is forwarded to the TLS-1.2 PRF, is the concatenation of the `ClientHello.Random + ServerHello.Random`.
- The label is `"master secret"` or `"extended master secret"`.

10.6.2 Input step types

psa_key_derivation_step_t (type)

Encoding of the step of a key derivation.

```
typedef uint16_t psa_key_derivation_step_t;
```

PSA_KEY_DERIVATION_INPUT_SECRET (macro)

A secret input for key derivation.

```
#define PSA_KEY_DERIVATION_INPUT_SECRET /* implementation-defined value */
```

This is typically a key of type PSA_KEY_TYPE_DERIVE passed to psa_key_derivation_input_key(), or the shared secret resulting from a key agreement obtained via psa_key_derivation_key_agreement().

The secret can also be a direct input passed to psa_key_derivation_input_bytes(). In this case, the derivation operation cannot be used to derive keys: the operation will only allow psa_key_derivation_output_bytes(), not psa_key_derivation_output_key().

PSA_KEY_DERIVATION_INPUT_LABEL (macro)

A label for key derivation.

```
#define PSA_KEY_DERIVATION_INPUT_LABEL /* implementation-defined value */
```

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

PSA_KEY_DERIVATION_INPUT_CONTEXT (macro)

A context for key derivation.

```
#define PSA_KEY_DERIVATION_INPUT_CONTEXT /* implementation-defined value */
```

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

PSA_KEY_DERIVATION_INPUT_SALT (macro)

A salt for key derivation.

```
#define PSA_KEY_DERIVATION_INPUT_SALT /* implementation-defined value */
```

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

PSA_KEY_DERIVATION_INPUT_INFO (macro)

An information string for key derivation.

```
#define PSA_KEY_DERIVATION_INPUT_INFO /* implementation-defined value */
```

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

PSA_KEY_DERIVATION_INPUT_SEED (macro)

A seed for key derivation.

```
#define PSA_KEY_DERIVATION_INPUT_SEED /* implementation-defined value */
```

This is typically a direct input. It can also be a key of type PSA_KEY_TYPE_RAW_DATA.

10.6.3 Key derivation functions

psa_key_derivation_operation_t (type)

The type of the state object for key derivation operations.

```
typedef /* implementation-defined type */ psa_key_derivation_operation_t;
```

Before calling any function on a key derivation operation object, the application must initialize it by any of the following means:

- Set the object to all-bits-zero, for example:

  ```
  psa_key_derivation_operation_t operation;
  memset(&operation, 0, sizeof(operation));
  ```

- Initialize the object to logical zero values by declaring the object as static or global without an explicit initializer, for example:

  ```
  static psa_key_derivation_operation_t operation;
  ```

- Initialize the object to the initializer PSA_KEY_DERIVATION_OPERATION_INIT, for example:

  ```
  psa_key_derivation_operation_t operation = PSA_KEY_DERIVATION_OPERATION_INIT;
  ```

- Assign the result of the function psa_key_derivation_operation_init() to the object, for example:

  ```
  psa_key_derivation_operation_t operation;
  operation = psa_key_derivation_operation_init();
  ```

This is an implementation-defined type. Applications that make assumptions about the content of this object will result in in implementation-specific behavior, and are non-portable.

PSA_KEY_DERIVATION_OPERATION_INIT (macro)

This macro returns a suitable initializer for a key derivation operation object of type psa_key_derivation_operation_t.

```
#define PSA_KEY_DERIVATION_OPERATION_INIT /* implementation-defined value */
```

psa_key_derivation_operation_init (function)

Return an initial value for a key derivation operation object.

```
psa_key_derivation_operation_t psa_key_derivation_operation_init(void);
```

Returns: `psa_key_derivation_operation_t`

`psa_key_derivation_setup` (function)

Set up a key derivation operation.

```
psa_status_t psa_key_derivation_setup(psa_key_derivation_operation_t * operation,
                                      psa_algorithm_t alg);
```

Parameters

`operation`	The key derivation operation object to set up. It must have been initialized but not set up yet.
`alg`	The key derivation algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_KEY_DERIVATION(alg)` is true).

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_ARGUMENT`	`alg` is not a key derivation algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a key derivation algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be inactive.
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

A key derivation algorithm takes some inputs and uses them to generate a byte stream in a deterministic way. This byte stream can be used to produce keys and other cryptographic material.

To derive a key:

1. Start with an initialized object of type `psa_key_derivation_operation_t`.
2. Call `psa_key_derivation_setup()` to select the algorithm.
3. Provide the inputs for the key derivation by calling `psa_key_derivation_input_bytes()` or `psa_key_derivation_input_key()` as appropriate. Which inputs are needed, in what order, whether keys are permitted, and what type of keys depends on the algorithm.

4. Optionally set the operation's maximum capacity with psa_key_derivation_set_capacity(). This can be done before, in the middle of, or after providing inputs. For some algorithms, this step is mandatory because the output depends on the maximum capacity.
5. To derive a key, call psa_key_derivation_output_key(). To derive a byte string for a different purpose, call psa_key_derivation_output_bytes(). Successive calls to these functions use successive output bytes calculated by the key derivation algorithm.
6. Clean up the key derivation operation object with psa_key_derivation_abort().

If this function returns an error, the key derivation operation object is not changed.

If an error occurs at any step after a call to psa_key_derivation_setup(), the operation will need to be reset by a call to psa_key_derivation_abort().

Implementations must reject an attempt to derive a key of size `0`.

`psa_key_derivation_get_capacity` (function)

Retrieve the current capacity of a key derivation operation.

```
psa_status_t psa_key_derivation_get_capacity(const psa_key_derivation_operation_t * operation,
                                             size_t * capacity);
```

Parameters

`operation`	The operation to query.
`capacity`	On success, the capacity of the operation.

Returns: `psa_status_t`

PSA_SUCCESS	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active.
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

The capacity of a key derivation is the maximum number of bytes that it can return. Reading *N* bytes of output from a key derivation operation reduces its capacity by at least *N*. The capacity can be reduced by more than *N* in the following situations:

- Calling psa_key_derivation_output_key() can reduce the capacity by more than the key size, depending on the type of key being generated. See psa_key_derivation_output_key() for details of the key derivation process.
- When the psa_key_derivation_operation_t object is operating as a deterministic random bit generator (DBRG), which reduces capacity in whole blocks, even when less than a block is read.

`psa_key_derivation_set_capacity` (function)

Set the maximum capacity of a key derivation operation.

```
psa_status_t psa_key_derivation_set_capacity(psa_key_derivation_operation_t * operation,
                                             size_t capacity);
```

Parameters

`operation`	The key derivation operation object to modify.
`capacity`	The new capacity of the operation. It must be less or equal to the operation's current capacity.

Returns: `psa_status_t`

`PSA_SUCCESS`	
`PSA_ERROR_INVALID_ARGUMENT`	`capacity` is larger than the operation's current capacity. In this case, the operation object remains valid and its capacity remains unchanged.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be active.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

The capacity of a key derivation operation is the maximum number of bytes that the key derivation operation can return from this point onwards.

`psa_key_derivation_input_bytes` (function)

Provide an input for key derivation or key agreement.

```
psa_status_t psa_key_derivation_input_bytes(psa_key_derivation_operation_t * operation,
                                            psa_key_derivation_step_t step,
                                            const uint8_t * data,
                                            size_t data_length);
```

Parameters

`operation`	The key derivation operation object to use. It must have been set up with `psa_key_derivation_setup()` and must not have produced any output yet.
`step`	Which step the input data is for.
`data`	Input data to use.
`data_length`	Size of the `data` buffer in bytes.

Returns: `psa_status_t`

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_ARGUMENT	`step` is not compatible with the operation's algorithm.
PSA_ERROR_INVALID_ARGUMENT	`step` does not allow direct inputs.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The operation state is not valid for this input `step`. This can happen if the application provides a step out of order or repeats a step that may not be repeated.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

Which inputs are required and in what order depends on the algorithm. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function passes direct inputs, which is usually correct for non-secret inputs. To pass a secret input, which is normally in a key object, call `psa_key_derivation_input_key()` instead of this function. Refer to the documentation of individual step types (`PSA_KEY_DERIVATION_INPUT_xxx` values of type `psa_key_derivation_step_t`) for more information.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_key_derivation_abort()`.

`psa_key_derivation_input_key` (function)

Provide an input for key derivation in the form of a key.

```
psa_status_t psa_key_derivation_input_key(psa_key_derivation_operation_t * operation,
                                          psa_key_derivation_step_t step,
                                          psa_key_id_t key);
```

Parameters

`operation`	The key derivation operation object to use. It must have been set up with `psa_key_derivation_setup()` and must not have produced any output yet.
`step`	Which step the input data is for.
`key`	Identifier of the key. It must have an appropriate type for `step` and must allow the usage `PSA_KEY_USAGE_DERIVE`.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_DERIVE flag.
PSA_ERROR_INVALID_ARGUMENT	step is not compatible with the operation's algorithm.
PSA_ERROR_INVALID_ARGUMENT	step does not allow key inputs of the given type or does not allow key inputs at all.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The operation state is not valid for this input step. This can happen if the application provides a step out of order or repeats a step that may not be repeated.
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Which inputs are required and in what order depends on the algorithm. Refer to the documentation of each key derivation or key agreement algorithm for information.

This function obtains input from a key object, which is usually correct for secret inputs or for non-secret personalization strings kept in the key store. To pass a non-secret parameter which is not in the key store, call psa_key_derivation_input_bytes() instead of this function. Refer to the documentation of individual step types (PSA_KEY_DERIVATION_INPUT_xxx values of type psa_key_derivation_step_t) for more information.

If this function returns an error status, the operation enters an error state and must be aborted by calling psa_key_derivation_abort().

psa_key_derivation_output_bytes (function)

Read some data from a key derivation operation.

```
psa_status_t psa_key_derivation_output_bytes(psa_key_derivation_operation_t * operation,
                                             uint8_t * output,
                                             size_t output_length);
```

Parameters

operation	The key derivation operation object to read from.
output	Buffer where the output will be written.

output_length	Number of bytes to output.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_INSUFFICIENT_DATA	The operation's capacity was less than output_length bytes. Note that in this case, no output is written to the output buffer. The operation's capacity is set to 0, thus subsequent calls to this function will not succeed, even with a smaller output buffer.
PSA_ERROR_BAD_STATE	The operation state is not valid: it must be active and completed all required input steps.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function calculates output bytes from a key derivation algorithm and returns those bytes. If the key derivation's output is viewed as a stream of bytes, this function consumes the requested number of bytes from the stream and returns them to the caller. The operation's capacity decreases by the number of bytes read.

If this function returns an error status other than PSA_ERROR_INSUFFICIENT_DATA, the operation enters an error state and must be aborted by calling psa_key_derivation_abort().

psa_key_derivation_output_key (function)

Derive a key from an ongoing key derivation operation.

```
psa_status_t psa_key_derivation_output_key(const psa_key_attributes_t * attributes,
                                           psa_key_derivation_operation_t * operation,
                                           psa_key_id_t * key);
```

Parameters

attributes — The attributes for the new key. This function uses the attributes as follows:

- The key type is required. It cannot be an asymmetric public key.
- The key size is required. It must be a valid size for the key type.

- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see *Permitted algorithms* on page 78.
- The key usage flags define what operations are permitted with the key, see *Key usage flags* on page 80.
- The key lifetime and identifier are required for a persistent key.

> **Note:**
>
> This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling `psa_get_key_attributes()` with the key's identifier.

`operation`	The key derivation operation object to read from.
`key`	On success, an identifier for the newly created key. `PSA_KEY_ID_NULL` on failure.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
`PSA_ERROR_ALREADY_EXISTS`	This is an attempt to create a persistent key, and there is already a persistent key with the given identifier.
`PSA_ERROR_INSUFFICIENT_DATA`	There was not enough data to create the desired key. Note that in this case, no output is written to the output buffer. The operation's capacity is set to `0`, thus subsequent calls to this function will not succeed, even with a smaller output buffer.
`PSA_ERROR_NOT_SUPPORTED`	The key type or key size is not supported, either by the implementation in general or in this particular location.
`PSA_ERROR_INVALID_ARGUMENT`	The key attributes, as a whole, are invalid.
`PSA_ERROR_INVALID_ARGUMENT`	The key type is an asymmetric public key type.
`PSA_ERROR_INVALID_ARGUMENT`	The key size is not a valid size for the key type.
`PSA_ERROR_NOT_PERMITTED`	The `PSA_KEY_DERIVATION_INPUT_SECRET` input was neither provided through a key nor the result of a key agreement.
`PSA_ERROR_BAD_STATE`	The operation state is not valid: it must be active and completed all required input steps.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	

PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

This function calculates output bytes from a key derivation algorithm and uses those bytes to generate a key deterministically. The key's location, policy, type and size are taken from attributes.

If the key derivation's output is viewed as a stream of bytes, this function consumes the required number of bytes from the stream. The operation's capacity decreases by the number of bytes used to derive the key.

If this function returns an error status other than PSA_ERROR_INSUFFICIENT_DATA, the operation enters an error state and must be aborted by calling psa_key_derivation_abort().

How much output is produced and consumed from the operation, and how the key is derived, depends on the key type. Table 6 describes the required key derivation procedures for standard key derivation algorithms. Implementations can use other methods for implementation-specific algorithms.

In all cases, the data that is read is discarded from the operation. The operation's capacity is decreased by the number of bytes read.

Table 6 Standard key derivation process

Key type	Key type details and derivation procedure
AES	PSA_KEY_TYPE_AES
ARC4	PSA_KEY_TYPE_ARC4
CAMELLIA	PSA_KEY_TYPE_CAMELLIA
ChaCha20	PSA_KEY_TYPE_CHACHA20
SM4	PSA_KEY_TYPE_SM4
Secrets for derivation	PSA_KEY_TYPE_DERIVE
HMAC	PSA_KEY_TYPE_HMAC For key types for which the key is an arbitrary sequence of bytes of a given size, this function is functionally equivalent to calling psa_key_derivation_output_bytes() and passing the resulting output to psa_import_key(). However, this function has a security benefit: if the implementation provides an isolation boundary then the key material is not exposed outside the isolation boundary. As a consequence, for these key types, this function always consumes exactly (bits/8) bytes from the operation.

Table 6 (continued)

Key type	Key type details and derivation procedure
DES	PSA_KEY_TYPE_DES, 64 bits. This function generates a key using the following process: 1. Draw an 8-byte string. 2. Set/clear the parity bits in each byte. 3. If the result is a forbidden weak key, discard the result and return to step 1. 4. Output the string.
2-key 3DES 3-key 3DES	PSA_KEY_TYPE_DES, 192 bits. PSA_KEY_TYPE_DES, 128 bits. The two or three keys are generated by repeated application of the process used to generate a DES key. For example, for 3-key 3DES, if the first 8 bytes specify a weak key and the next 8 bytes do not, discard the first 8 bytes, use the next 8 bytes as the first key, and continue reading output from the operation to derive the other two keys.

Table 6 (continued)

Key type	Key type details and derivation procedure
Finite-field Diffie-Hellman keys	`PSA_KEY_TYPE_DH_KEY_PAIR(dh_family)` where `dh_family` designates any Diffie-Hellman family.
ECC keys on a Weierstrass elliptic curve	`PSA_KEY_TYPE_ECC_KEY_PAIR(ecc_family)` where `ecc_family` designates a Weierstrass curve family. These key types require the generation of a private key which is an integer in the range [1, *N* - 1], where *N* is the boundary of the private key domain: *N* is the prime *p* for Diffie-Hellman, or the order of the curve's base point for ECC. Let `m` be the bit size of *N*, such that `2^m` > *N* >= `2^(m-1)`. This function generates the private key using the following process: 1. Draw a byte string of length `ceiling(m/8)` bytes. 2. If `m` is not a multiple of `8`, set the most significant `(8 * ceiling(m/8) - m)` bits of the first byte in the string to zero. 3. Convert the string to integer *k* by decoding it as a big-endian byte string. 4. If *k* > *N* - `2`, discard the result and return to step 1. 5. Output *k* + `1` as the private key. This method allows compliance to NIST standards, specifically the methods titled *Key-Pair Generation by Testing Candidates* in the following publications: • *NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography* [SP800-56A] §5.6.1.1.4 for Diffie-Hellman keys. • [SP800-56A] §5.6.1.2.2 or *FIPS Publication 186-4: Digital Signature Standard (DSS)* [FIPS186-4] §B.4.2 for elliptic curve keys.
ECC keys on a Montgomery elliptic curve	`PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_MONTGOMERY)` This function always draws a byte string whose length is determined by the curve, and sets the mandatory bits accordingly. That is: • Curve25519 (`PSA_ECC_FAMILY_MONTGOMERY`, 255 bits): draw a 32-byte string and process it as specified in *Elliptic Curves for Security* [RFC7748] §5. • Curve448 (`PSA_ECC_FAMILY_MONTGOMERY`, 448 bits): draw a 56-byte string and process it as specified in [RFC7748] §5.
Other key types	This includes `PSA_KEY_TYPE_RSA_KEY_PAIR`. The way in which the operation output is consumed is implementation-defined.

For algorithms that take an input step PSA_KEY_DERIVATION_INPUT_SECRET, the input to that step must be provided with psa_key_derivation_input_key(). Future versions of this specification might include additional restrictions on the derived key based on the attributes and strength of the secret key.

psa_key_derivation_abort (function)

Abort a key derivation operation.

```
psa_status_t psa_key_derivation_abort(psa_key_derivation_operation_t * operation);
```

Parameters

operation	The operation to abort.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Aborting an operation frees all associated resources except for the operation object itself. Once aborted, the operation object can be reused for another operation by calling psa_key_derivation_setup() again.

This function can be called at any time after the operation object has been initialized as described in psa_key_derivation_operation_t.

In particular, it is valid to call psa_key_derivation_abort() twice, or to call psa_key_derivation_abort() on an operation that has not been set up.

10.6.4 Support macros

PSA_ALG_IS_HKDF (macro)

Whether the specified algorithm is an HKDF algorithm.

```
#define PSA_ALG_IS_HKDF(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is an HKDF algorithm, 0 otherwise. This macro can return either 0 or 1 if alg is not a supported key derivation algorithm identifier.

Description

HKDF is a family of key derivation algorithms that are based on a hash function and the HMAC construction.

PSA_ALG_IS_TLS12_PRF (macro)

Whether the specified algorithm is a TLS-1.2 PRF algorithm.

```
#define PSA_ALG_IS_TLS12_PRF(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a TLS-1.2 PRF algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported key derivation algorithm identifier.

PSA_ALG_IS_TLS12_PSK_TO_MS (macro)

Whether the specified algorithm is a TLS-1.2 PSK to MS algorithm.

```
#define PSA_ALG_IS_TLS12_PSK_TO_MS(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a TLS-1.2 PSK to MS algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported key derivation algorithm identifier.

PSA_KEY_DERIVATION_UNLIMITED_CAPACITY (macro)

Use the maximum possible capacity for a key derivation operation.

```
#define PSA_KEY_DERIVATION_UNLIMITED_CAPACITY \
    /* implementation-defined value */
```

Use this value as the capacity argument when setting up a key derivation to specify that the operation will use the maximum possible capacity. The value of the maximum possible capacity depends on the key derivation algorithm.

PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE (macro)

This macro returns the maximum supported length of the PSK for the TLS-1.2 PSK-to-MS key derivation.

```
#define PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE /* implementation-defined value */
```

This implementation-defined value specifies the maximum length for the PSK input used with a `PSA_ALG_TLS12_PSK_TO_MS()` key agreement algorithm.

Quoting *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)* [RFC4279] §5.3:

> TLS implementations supporting these cipher suites MUST support arbitrary PSK identities up to 128 octets in length, and arbitrary PSKs up to 64 octets in length. Supporting longer identities and keys is RECOMMENDED.

Therefore, it is recommended that implementations define `PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE` with a value greater than or equal to `64`.

10.7 Asymmetric signature

10.7.1 Asymmetric signature algorithms

PSA_ALG_RSA_PKCS1V15_SIGN (macro)

The RSA PKCS#1 v1.5 message signature scheme, with hashing.

```
#define PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true). This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding RSA PKCS#1 v1.5 signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This signature scheme is defined by *PKCS #1: RSA Cryptography Specifications Version 2.2* [RFC8017] §8.2 under the name RSASSA-PKCS1-v1_5.

When used with `psa_sign_hash()` or `psa_verify_hash()`, the provided `hash` parameter is used as *H* from step 2 onwards in the message encoding algorithm `EMSA-PKCS1-V1_5-ENCODE()` in [RFC8017] §9.2. *H* is usually the message digest, using the `hash_alg` hash algorithm.

PSA_ALG_RSA_PKCS1V15_SIGN_RAW (macro)

The raw RSA PKCS#1 v1.5 signature algorithm, without hashing.

```
#define PSA_ALG_RSA_PKCS1V15_SIGN_RAW ((psa_algorithm_t) 0x06000200)
```

This algorithm can be only used with the `psa_sign_hash()` and `psa_verify_hash()` functions.

This signature scheme is defined by *PKCS #1: RSA Cryptography Specifications Version 2.2* [RFC8017] §8.2 under the name RSASSA-PKCS1-v1_5.

The `hash` parameter to `psa_sign_hash()` or `psa_verify_hash()` is used as *T* from step 3 onwards in the message encoding algorithm `EMSA-PKCS1-V1_5-ENCODE()` in [RFC8017] §9.2. *T* is the DER encoding of the *DigestInfo* structure normally produced by step 2 in the message encoding algorithm.

PSA_ALG_RSA_PSS (macro)

The RSA PSS message signature scheme, with hashing.

```
#define PSA_ALG_RSA_PSS(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true). This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding RSA PSS signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the signature scheme defined by [RFC8017] §8.1 under the name RSASSA-PSS, with the following options:

- The mask generation function is *MGF1* defined by [RFC8017] Appendix B.
- The salt length is equal to the length of the hash.
- The specified hash algorithm is used to hash the input message, to create the salted hash, and for the mask generation.

PSA_ALG_ECDSA (macro)

The randomized ECDSA signature scheme, with hashing.

```
#define PSA_ALG_ECDSA(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true). This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding randomized ECDSA signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

Note:

When based on the same hash algorithm, the verification operations for `PSA_ALG_ECDSA` and `PSA_ALG_DETERMINISTIC_ECDSA` are identical. A signature created using `PSA_ALG_ECDSA` can be verified with the same key using either `PSA_ALG_ECDSA` or `PSA_ALG_DETERMINISTIC_ECDSA`. Similarly, a signature created using `PSA_ALG_DETERMINISTIC_ECDSA` can be verified with the same key using either `PSA_ALG_ECDSA` or `PSA_ALG_DETERMINISTIC_ECDSA`.

In particular, it is impossible to determine whether a signature was produced with deterministic ECDSA or with randomized ECDSA: it is only possible to verify that a signature was made with ECDSA with the private key corresponding to the public key used for the verification.

This signature scheme is defined by *SEC 1: Elliptic Curve Cryptography* [SEC1], and also by *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)* [X9-62], with a random per-message secret number *k*.

The representation of the signature as a byte string consists of the concatenation of the signature values *r* and *s*. Each of *r* and *s* is encoded as an *N*-octet string, where *N* is the length of the base point of the curve in octets. Each value is represented in big-endian order, with the most significant octet first.

PSA_ALG_ECDSA_ANY (macro)

The randomized ECDSA signature scheme, without hashing.

```
#define PSA_ALG_ECDSA_ANY ((psa_algorithm_t) 0x06000600)
```

This algorithm can be only used with the `psa_sign_hash()` and `psa_verify_hash()` functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the same signature scheme as `PSA_ALG_ECDSA()`, but without specifying a hash algorithm, and skipping the message hashing operation.

This algorithm is only recommended to sign or verify a sequence of bytes that are an already-calculated hash. Note that the input is padded with zeros on the left or truncated on the right as required to fit the curve size.

PSA_ALG_DETERMINISTIC_ECDSA (macro)

Deterministic ECDSA signature scheme, with hashing.

```
#define PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true). This includes `PSA_ALG_ANY_HASH` when specifying the algorithm in a key policy.

Returns

The corresponding deterministic ECDSA signature algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This algorithm can be used with both the message and hash signature functions.

> **Note:**
>
> When based on the same hash algorithm, the verification operations for PSA_ALG_ECDSA and PSA_ALG_DETERMINISTIC_ECDSA are identical. A signature created using PSA_ALG_ECDSA can be verified with the same key using either PSA_ALG_ECDSA or PSA_ALG_DETERMINISTIC_ECDSA. Similarly, a signature created using PSA_ALG_DETERMINISTIC_ECDSA can be verified with the same key using either PSA_ALG_ECDSA or PSA_ALG_DETERMINISTIC_ECDSA.
>
> In particular, it is impossible to determine whether a signature was produced with deterministic ECDSA or with randomized ECDSA: it is only possible to verify that a signature was made with ECDSA with the private key corresponding to the public key used for the verification.

This is the deterministic ECDSA signature scheme defined by *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)* [RFC6979].

The representation of a signature is the same as with PSA_ALG_ECDSA().

10.7.2 Asymmetric signature functions

psa_sign_message (function)

Sign a message with a private key. For hash-and-sign algorithms, this includes the hashing step.

```
psa_status_t psa_sign_message(psa_key_id_t key,
                              psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
                              uint8_t * signature,
                              size_t signature_size,
                              size_t * signature_length);
```

Parameters

key	Identifier of the key to use for the operation. It must be an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_SIGN_MESSAGE.
alg	An asymmetric signature algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_MESSAGE(alg) is true), that is compatible with the type of key.
input	The input message to sign.
input_length	Size of the input buffer in bytes.
signature	Buffer where the signature is to be written.
signature_size	Size of the signature buffer in bytes. This must be appropriate for the selected algorithm and key:

- The required signature size is PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of key.
- PSA_SIGNATURE_MAX_SIZE evaluates to the maximum signature size of any supported signature algorithm.

signature_length — On success, the number of bytes that make up the returned signature value.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_SIGN_MESSAGE flag, or it does not permit the requested algorithm.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the signature buffer is too small. PSA_SIGN_OUTPUT_SIZE() or PSA_SIGNATURE_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Note:

To perform a multi-part hash-and-sign signature algorithm, first use a multi-part hash operation and then pass the resulting hash to psa_sign_hash(). PSA_ALG_GET_HASH(alg) can be used to determine the hash algorithm to use.

psa_verify_message (function)

Verify the signature of a message with a public key, using a hash-and-sign verification algorithm.

```
psa_status_t psa_verify_message(psa_key_id_t key,
                                psa_algorithm_t alg,
                                const uint8_t * input,
                                size_t input_length,
                                const uint8_t * signature,
                                size_t signature_length);
```

Parameters

key	Identifier of the key to use for the operation. It must be a public key or an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_VERIFY_MESSAGE.
alg	An asymmetric signature algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_MESSAGE(alg) is true), that is compatible with the type of key.
input	The message whose signature is to be verified.
input_length	Size of the input buffer in bytes.
signature	Buffer containing the signature to verify.
signature_length	Size of the signature buffer in bytes.

Returns: psa_status_t

PSA_SUCCESS	The signature is valid.
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_VERIFY_MESSAGE flag, or it does not permit the requested algorithm.
PSA_ERROR_INVALID_SIGNATURE	The calculation was performed successfully, but the passed signature is not a valid signature.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Note:

To perform a multi-part hash-and-sign signature verification algorithm, first use a multi-part hash operation to hash the message and then pass the resulting hash to psa_verify_hash(). PSA_ALG_GET_HASH(alg) can be used to determine the hash algorithm to use.

psa_sign_hash (function)

Sign an already-calculated hash with a private key.

```
psa_status_t psa_sign_hash(psa_key_id_t key,
                           psa_algorithm_t alg,
                           const uint8_t * hash,
                           size_t hash_length,
                           uint8_t * signature,
                           size_t signature_size,
                           size_t * signature_length);
```

Parameters

key	Identifier of the key to use for the operation. It must be an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_SIGN_HASH.
alg	An asymmetric signature algorithm that separates the hash and sign operations (PSA_ALG_XXX value such that PSA_ALG_IS_SIGN_HASH(alg) is true), that is compatible with the type of key.
hash	The input to sign. This is usually the hash of a message. See the detailed description of this function and the description of individual signature algorithms for a detailed description of acceptable inputs.
hash_length	Size of the hash buffer in bytes.
signature	Buffer where the signature is to be written.
signature_size	Size of the signature buffer in bytes. This must be appropriate for the selected algorithm and key: • The required signature size is PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) where key_type and key_bits are the type and bit-size respectively of key. • PSA_SIGNATURE_MAX_SIZE evaluates to the maximum signature size of any supported signature algorithm.
signature_length	On success, the number of bytes that make up the returned signature value.

Returns: psa_status_t

PSA_SUCCESS	
PSA_ERROR_INVALID_HANDLE	
PSA_ERROR_NOT_PERMITTED	The key does not have the PSA_KEY_USAGE_SIGN_HASH flag, or it does not permit the requested algorithm.
PSA_ERROR_BUFFER_TOO_SMALL	The size of the signature buffer is too small. PSA_SIGN_OUTPUT_SIZE()

	or PSA_SIGNATURE_MAX_SIZE can be used to determine the required buffer size.
PSA_ERROR_NOT_SUPPORTED	
PSA_ERROR_INVALID_ARGUMENT	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_HARDWARE_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

With most signature mechanisms that follow the hash-and-sign paradigm, the hash input to this function is the hash of the message to sign. The hash algorithm is encoded in the signature algorithm.

Some hash-and-sign mechanisms apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. The current version of this specification defines one such signature algorithm: PSA_ALG_RSA_PKCS1V15_SIGN_RAW.

> **Note:**
>
> To perform a hash-and-sign algorithm, the hash must be calculated before passing it to this function. This can be done by calling psa_hash_compute() or with a multi-part hash operation. Alternatively, to hash and sign a message in a single call, use psa_sign_message().

psa_verify_hash (function)

Verify the signature of a hash or short message using a public key.

```
psa_status_t psa_verify_hash(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const uint8_t * hash,
                             size_t hash_length,
                             const uint8_t * signature,
                             size_t signature_length);
```

Parameters

key	Identifier of the key to use for the operation. It must be a public key or an asymmetric key pair. The key must allow the usage PSA_KEY_USAGE_VERIFY_HASH.

`alg`	An asymmetric signature algorithm that separates the hash and sign operations (`PSA_ALG_XXX` value such that `PSA_ALG_IS_SIGN_HASH(alg)` is true), that is compatible with the type of `key`.
`hash`	The input whose signature is to be verified. This is usually the hash of a message. See the detailed description of this function and the description of individual signature algorithms for a detailed description of acceptable inputs.
`hash_length`	Size of the `hash` buffer in bytes.
`signature`	Buffer containing the signature to verify.
`signature_length`	Size of the `signature` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	The signature is valid.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_VERIFY_HASH` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_SIGNATURE`	The calculation was performed successfully, but the passed signature is not a valid signature.
`PSA_ERROR_NOT_SUPPORTED`	
`PSA_ERROR_INVALID_ARGUMENT`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

With most signature mechanisms that follow the hash-and-sign paradigm, the hash input to this function is the hash of the message to sign. The hash algorithm is encoded in the signature algorithm.

Some hash-and-sign mechanisms apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. The current version of this specification defines one such signature algorithm: `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`.

Note:

To perform a hash-and-sign verification algorithm, the hash must be calculated before passing it to this function. This can be done by calling `psa_hash_compute()` or with a multi-part hash operation.

Alternatively, to hash and verify a message signature in a single call, use psa_verify_message().

10.7.3 Support macros

PSA_ALG_IS_SIGN_MESSAGE (macro)

Whether the specified algorithm is a signature algorithm that can be used with psa_sign_message() and psa_verify_message().

```
#define PSA_ALG_IS_SIGN_MESSAGE(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is a signature algorithm that can be used to sign a message. 0 if alg is a signature algorithm that can only be used to sign an already-calculated hash. 0 if alg is not a signature algorithm. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

PSA_ALG_IS_SIGN_HASH (macro)

Whether the specified algorithm is a signature algorithm that can be used with psa_sign_hash() and psa_verify_hash().

```
#define PSA_ALG_IS_SIGN_HASH(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is a signature algorithm that can be used to sign a hash. 0 if alg is a signature algorithm that can only be used to sign a message. 0 if alg is not a signature algorithm. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

PSA_ALG_IS_RSA_PKCS1V15_SIGN (macro)

Whether the specified algorithm is an RSA PKCS#1 v1.5 signature algorithm.

```
#define PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) /* specification-defined value */
```

Parameters

alg	An algorithm identifier (value of type psa_algorithm_t).

Returns

1 if alg is an RSA PKCS#1 v1.5 signature algorithm, 0 otherwise.

This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

PSA_ALG_IS_RSA_PSS (macro)

Whether the specified algorithm is an RSA PSS signature algorithm.

```
#define PSA_ALG_IS_RSA_PSS(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is an RSA PSS signature algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

PSA_ALG_IS_ECDSA (macro)

Whether the specified algorithm is ECDSA.

```
#define PSA_ALG_IS_ECDSA(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is an ECDSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

PSA_ALG_IS_DETERMINISTIC_ECDSA (macro)

Whether the specified algorithm is deterministic ECDSA.

```
#define PSA_ALG_IS_DETERMINISTIC_ECDSA(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a deterministic ECDSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See also `PSA_ALG_IS_ECDSA()` and `PSA_ALG_IS_RANDOMIZED_ECDSA()`.

PSA_ALG_IS_RANDOMIZED_ECDSA (macro)

Whether the specified algorithm is randomized ECDSA.

```
#define PSA_ALG_IS_RANDOMIZED_ECDSA(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a randomized ECDSA algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

See also `PSA_ALG_IS_ECDSA()` and `PSA_ALG_IS_DETERMINISTIC_ECDSA()`.

PSA_ALG_IS_HASH_AND_SIGN (macro)

Whether the specified algorithm is a hash-and-sign algorithm that signs exactly the hash value.

```
#define PSA_ALG_IS_HASH_AND_SIGN(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a hash-and-sign algorithm that signs exactly the hash value, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

This macro identifies algorithms that can be used with `psa_sign_hash()` that use the exact message hash value as an input the signature operation. This excludes hash-and-sign algorithms that require a encoded or modified hash for the signature step in the algorithm, such as `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`.

PSA_ALG_ANY_HASH (macro)

When setting a hash-and-sign algorithm in a key policy, permit any hash algorithm.

```
#define PSA_ALG_ANY_HASH ((psa_algorithm_t)0x020000ff)
```

This value can be used to form the permitted algorithm attribute of a key policy for a signature algorithm that is parametrized by a hash. A key with this policy can then be used to perform operations using the same signature algorithm parametrized with any supported hash. A signature algorithm created using this macro is a wildcard algorithm, and `PSA_ALG_IS_WILDCARD()` will return true.

This value must not be used to build other algorithms that are parametrized over a hash. For any valid use of this macro to build an algorithm `alg`, `PSA_ALG_IS_HASH_AND_SIGN(alg)` is true.

This value must not be used to build an algorithm specification to perform an operation. It is only valid for setting the permitted algorithm in a key policy.

Usage

For example, suppose that PSA_xxx_SIGNATURE is one of the following macros:

- PSA_ALG_RSA_PKCS1V15_SIGN
- PSA_ALG_RSA_PSS
- PSA_ALG_ECDSA
- PSA_ALG_DETERMINISTIC_ECDSA

The following sequence of operations shows how PSA_ALG_ANY_HASH can be used in a key policy:

1. Set the key usage flags using PSA_ALG_ANY_HASH, for example:

   ```
   psa_set_key_usage_flags(&attributes, PSA_KEY_USAGE_SIGN_MESSAGE); // or VERIFY_MESSAGE
   psa_set_key_algorithm(&attributes, PSA_xxx_SIGNATURE(PSA_ALG_ANY_HASH));
   ```

2. Import or generate key material.
3. Call psa_sign_message() or psa_verify_message(), passing an algorithm built from PSA_xxx_SIGNATURE and a specific hash. Each call to sign or verify a message can use a different hash algorithm.

   ```
   psa_sign_message(key, PSA_xxx_SIGNATURE(PSA_ALG_SHA_256), ...);
   psa_sign_message(key, PSA_xxx_SIGNATURE(PSA_ALG_SHA_512), ...);
   psa_sign_message(key, PSA_xxx_SIGNATURE(PSA_ALG_SHA3_256), ...);
   ```

PSA_SIGN_OUTPUT_SIZE (macro)

Sufficient signature buffer size for psa_sign_message() and psa_sign_hash().

```
#define PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
```

Parameters

key_type	An asymmetric key type. This can be a key pair type or a public key type.
key_bits	The size of the key in bits.
alg	The signature algorithm.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that psa_sign_message() and psa_sign_hash() will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

Description

This macro returns a sufficient buffer size for a signature using a key of the specified type and size, with the specified algorithm. Note that the actual size of the signature might be smaller, as some algorithms produce a variable-size signature.

Warning: This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also `PSA_SIGNATURE_MAX_SIZE`.

PSA_SIGNATURE_MAX_SIZE (macro)

Maximum size of an asymmetric signature.

```
#define PSA_SIGNATURE_MAX_SIZE /* implementation-defined value */
```

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of an asymmetric signature supported by the implementation, in bytes. The value must not be smaller than this maximum.

See also `PSA_SIGN_OUTPUT_SIZE()`.

10.8 Asymmetric encryption

10.8.1 Asymmetric encryption algorithms

PSA_ALG_RSA_PKCS1V15_CRYPT (macro)

The RSA PKCS#1 v1.5 asymmetric encryption algorithm.

```
#define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x07000200)
```

This encryption scheme is defined by *PKCS #1: RSA Cryptography Specifications Version 2.2* [RFC8017] §7.2 under the name RSAES-PKCS-v1_5.

PSA_ALG_RSA_OAEP (macro)

The RSA OAEP asymmetric encryption algorithm.

```
#define PSA_ALG_RSA_OAEP(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	The hash algorithm (`PSA_ALG_XXX` value such that `PSA_ALG_IS_HASH(hash_alg)` is true) to use for *MGF1*.

Returns

The corresponding RSA OAEP encryption algorithm.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This encryption scheme is defined by [RFC8017] §7.1 under the name RSAES-OAEP, with the mask generation function *MGF1* defined in [RFC8017] Appendix B.

10.8.2 Asymmetric encryption functions

`psa_asymmetric_encrypt` (function)

Encrypt a short message with a public key.

```
psa_status_t psa_asymmetric_encrypt(psa_key_id_t key,
                                    psa_algorithm_t alg,
                                    const uint8_t * input,
                                    size_t input_length,
                                    const uint8_t * salt,
                                    size_t salt_length,
                                    uint8_t * output,
                                    size_t output_size,
                                    size_t * output_length);
```

Parameters

`key`	Identifer of the key to use for the operation. It must be a public key or an asymmetric key pair. It must allow the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	An asymmetric encryption algorithm that is compatible with the type of `key`.
`input`	The message to encrypt.
`input_length`	Size of the `input` buffer in bytes.
`salt`	A salt or label, if supported by the encryption algorithm. If the algorithm does not support a salt, pass `NULL`. If the algorithm supports an optional salt, pass `NULL` to indicate that there is no salt.
`salt_length`	Size of the `salt` buffer in bytes. If `salt` is `NULL`, pass `0`.
`output`	Buffer where the encrypted message is to be written.
`output_size`	Size of the `output` buffer in bytes. This must be appropriate for the selected algorithm and key: • The required output size is `PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg)` where `key_type` and `key_bits` are the type and bit-size respectively of `key`. • `PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported asymmetric encryption.
`output_length`	On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_ENCRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `output` buffer is too small. `PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE()` or

PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE can be used to determine the required buffer size.

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INVALID_ARGUMENT

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_BAD_STATE — The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

- For PSA_ALG_RSA_PKCS1V15_CRYPT, no salt is supported.

psa_asymmetric_decrypt (function)

Decrypt a short message with a private key.

```
psa_status_t psa_asymmetric_decrypt(psa_key_id_t key,
                                    psa_algorithm_t alg,
                                    const uint8_t * input,
                                    size_t input_length,
                                    const uint8_t * salt,
                                    size_t salt_length,
                                    uint8_t * output,
                                    size_t output_size,
                                    size_t * output_length);
```

Parameters

key	Identifier of the key to use for the operation. It must be an asymmetric key pair. It must allow the usage PSA_KEY_USAGE_DECRYPT.
alg	An asymmetric encryption algorithm that is compatible with the type of key.
input	The message to decrypt.
input_length	Size of the input buffer in bytes.
salt	A salt or label, if supported by the encryption algorithm. If the algorithm does not support a salt, pass NULL. If the algorithm supports an optional salt, pass NULL to indicate that there is no salt.
salt_length	Size of the salt buffer in bytes. If salt is NULL, pass 0.

`output`	Buffer where the decrypted message is to be written.
`output_size`	Size of the `output` buffer in bytes. This must be appropriate for the selected algorithm and key: • The required output size is `PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg)` where `key_type` and `key_bits` are the type and bit-size respectively of `key`. • `PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported asymmetric decryption.
`output_length`	On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_DECRYPT` flag, or it does not permit the requested algorithm.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `output` buffer is too small. `PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE()` or `PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE` can be used to determine the required buffer size.
`PSA_ERROR_NOT_SUPPORTED`	
`PSA_ERROR_INVALID_ARGUMENT`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INVALID_PADDING`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

- For `PSA_ALG_RSA_PKCS1V15_CRYPT`, no salt is supported.

10.8.3 Support macros

PSA_ALG_IS_RSA_OAEP (macro)

Whether the specified algorithm is an RSA OAEP encryption algorithm.

```
#define PSA_ALG_IS_RSA_OAEP(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is an RSA OAEP algorithm, `0` otherwise.

This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE (macro)

Sufficient output buffer size for `psa_asymmetric_encrypt()`.

```
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
```

Parameters

`key_type`	An asymmetric key type, either a key pair or a public key.
`key_bits`	The size of the key in bits.
`alg`	The asymmetric encryption algorithm.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that `psa_asymmetric_encrypt()` will not fail with `PSA_ERROR_BUFFER_TOO_SMALL`. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or `0`. If the parameters are not valid, the return value is unspecified.

Description

This macro returns a sufficient buffer size for a ciphertext produced using a key of the specified type and size, with the specified algorithm. Note that the actual size of the ciphertext might be smaller, depending on the algorithm.

> **Warning:** This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also `PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE`.

PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for `psa_asymmetric_encrypt()`, for any supported asymmetric encryption.

```
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
```

See also PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE().

PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE (macro)

Sufficient output buffer size for psa_asymmetric_decrypt().

```
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
```

Parameters

key_type	An asymmetric key type, either a key pair or a public key.
key_bits	The size of the key in bits.
alg	The asymmetric encryption algorithm.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that psa_asymmetric_decrypt() will not fail with PSA_ERROR_BUFFER_TOO_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

Description

This macro returns a sufficient buffer size for a plaintext produced using a key of the specified type and size, with the specified algorithm. Note that the actual size of the plaintext might be smaller, depending on the algorithm.

> **Warning:** This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE.

PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for psa_asymmetric_decrypt(), for any supported asymmetric decryption.

```
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
```

See also PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE().

10.9 Key agreement

10.9.1 Key agreement algorithms

PSA_ALG_KEY_AGREEMENT (macro)

Macro to build a combined algorithm that chains a key agreement with a key derivation.

```
#define PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg) \
    /* specification-defined value */
```

Parameters

ka_alg	A key agreement algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_AGREEMENT(ka_alg) is true).
kdf_alg	A key derivation algorithm (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_DERIVATION(kdf_alg) is true).

Returns

The corresponding key agreement and derivation algorithm.

Unspecified if ka_alg is not a supported key agreement algorithm or kdf_alg is not a supported key derivation algorithm.

Description

A combined key agreement algorithm is used with a multi-part key derivation operation, using a call to psa_key_derivation_key_agreement().

The component parts of a key agreement algorithm can be extracted using PSA_ALG_KEY_AGREEMENT_GET_BASE() and PSA_ALG_KEY_AGREEMENT_GET_KDF().

PSA_ALG_FFDH (macro)

The finite-field Diffie-Hellman (DH) key agreement algorithm.

```
#define PSA_ALG_FFDH ((psa_algorithm_t)0x09010000)
```

This algorithm can be used directly in a call to psa_raw_key_agreement(), or combined with a key derivation operation using PSA_ALG_KEY_AGREEMENT() for use with psa_key_derivation_key_agreement().

When used as part of a multi-part key derivation operation, this implements a Diffie-Hellman key agreement scheme using a single Diffie-Hellman key-pair for each participant. This includes the *dhEphem*, *dhOneFlow*, and *dhStatic* schemes. The input step PSA_KEY_DERIVATION_INPUT_SECRET is used when providing the secret and peer keys to the operation.

The shared secret produced by this key agreement algorithm is g^{ab} in big-endian format. It is ceiling(m / 8) bytes long where m is the size of the prime p in bits.

This key agreement scheme is defined by *NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography* [SP800-56A] §5.7.1.1 under the name FFC DH.

PSA_ALG_ECDH (macro)

The elliptic curve Diffie-Hellman (ECDH) key agreement algorithm.

```
#define PSA_ALG_ECDH ((psa_algorithm_t)0x09020000)
```

This algorithm can be used directly in a call to `psa_raw_key_agreement()`, or combined with a key derivation operation using `PSA_ALG_KEY_AGREEMENT()` for use with `psa_key_derivation_key_agreement()`.

When used as part of a multi-part key derivation operation, this implements a Diffie-Hellman key agreement scheme using a single elliptic curve key-pair for each participant. This includes the *Ephemeral unified model*, the *Static unified model*, and the *One-pass Diffie-Hellman* schemes. The input step `PSA_KEY_DERIVATION_INPUT_SECRET` is used when providing the secret and peer keys to the operation.

The shared secret produced by key agreement is the x-coordinate of the shared secret point. It is always `ceiling(m / 8)` bytes long where `m` is the bit size associated with the curve, i.e. the bit size of the order of the curve's coordinate field. When `m` is not a multiple of 8, the byte containing the most significant bit of the shared secret is padded with zero bits. The byte order is either little-endian or big-endian depending on the curve type.

- For Montgomery curves (curve family `PSA_ECC_FAMILY_MONTGOMERY`), the shared secret is the x-coordinate of `Z = d_A Q_B = d_B Q_A` in little-endian byte order.
  - For Curve25519, this is the X25519 function defined in *Curve25519: new Diffie-Hellman speed records* [Curve25519]. The bit size `m` is 255.
  - For Curve448, this is the X448 function defined in *Ed448-Goldilocks, a new elliptic curve* [Curve448]. The bit size `m` is 448.
- For Weierstrass curves (curve families `PSA_ECC_FAMILY_SECP_XX`, `PSA_ECC_FAMILY_SECT_XX`, `PSA_ECC_FAMILY_BRAINPOOL_P_R1` and `PSA_ECC_FAMILY_FRP`) the shared secret is the x-coordinate of `Z = h d_A Q_B = h d_B Q_A` in big-endian byte order. This is the Elliptic Curve Cryptography Cofactor Diffie-Hellman primitive defined by *SEC 1: Elliptic Curve Cryptography* [SEC1] §3.3.2 as, and also as ECC CDH by *NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography* [SP800-56A] §5.7.1.2.
  - Over prime fields (curve families `PSA_ECC_FAMILY_SECP_XX`, `PSA_ECC_FAMILY_BRAINPOOL_P_R1` and `PSA_ECC_FAMILY_FRP`), the bit size is `m = ceiling(log_2(p))` for the field `F_p`.
  - Over binary fields (curve families `PSA_ECC_FAMILY_SECT_XX`), the bit size is `m` for the field `F_{2^m}`.

> **Note:**
> The cofactor Diffie-Hellman primitive is equivalent to the standard elliptic curve Diffie-Hellman calculation `Z = d_A Q_B = d_B Q_A` ([SEC1] §3.3.1) for curves where the cofactor `h` is `1`. This is true for all curves in the `PSA_ECC_FAMILY_SECP_XX`, `PSA_ECC_FAMILY_BRAINPOOL_P_R1`, and `PSA_ECC_FAMILY_FRP` families.

10.9.2 Standalone key agreement

psa_raw_key_agreement (function)

Perform a key agreement and return the raw shared secret.

```
psa_status_t psa_raw_key_agreement(psa_algorithm_t alg,
                                   psa_key_id_t private_key,
                                   const uint8_t * peer_key,
                                   size_t peer_key_length,
                                   uint8_t * output,
                                   size_t output_size,
                                   size_t * output_length);
```

Parameters

`alg`	The key agreement algorithm to compute (`PSA_ALG_XXX` value such that `PSA_ALG_IS_RAW_KEY_AGREEMENT(alg)` is true).
`private_key`	Identifier of the private key to use. It must allow the usage `PSA_KEY_USAGE_DERIVE`.
`peer_key`	Public key of the peer. It must be in the same format that `psa_import_key()` accepts. The standard formats for public keys are documented in the documentation of `psa_export_public_key()`.
`peer_key_length`	Size of `peer_key` in bytes.
`output`	Buffer where the raw shared secret is to be written.
`output_size`	Size of the `output` buffer in bytes. This must be appropriate for the keys: • The required output size is `PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(type, bits)` where `type` is the type of `private_key` and `bits` is the bit-size of either `private_key` or the `peer_key`. • `PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported raw key agreement algorithm.
`output_length`	On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`alg` is not a key agreement algorithm
`PSA_ERROR_INVALID_ARGUMENT`	`private_key` is not compatible with `alg`, or `peer_key` is not valid for `alg` or not compatible with `private_key`.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `output` buffer is too small. `PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE()` or `PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE` can be used to determine the required buffer size.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not a supported key agreement algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

PSA_ERROR_BAD_STATE — The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Warning: The raw result of a key agreement algorithm such as finite-field Diffie-Hellman or elliptic curve Diffie-Hellman has biases, and is not suitable for use as key material. Instead it is recommended that the result is used as input to a key derivation algorithm. To chain a key agreement with a key derivation, use psa_key_derivation_key_agreement() and other functions from the key derivation interface.

10.9.3 Combining key agreement and key derivation

psa_key_derivation_key_agreement (function)

Perform a key agreement and use the shared secret as input to a key derivation.

```
psa_status_t psa_key_derivation_key_agreement(psa_key_derivation_operation_t * operation,
                                              psa_key_derivation_step_t step,
                                              psa_key_id_t private_key,
                                              const uint8_t * peer_key,
                                              size_t peer_key_length);
```

Parameters

operation — The key derivation operation object to use. It must have been set up with psa_key_derivation_setup() with a key agreement and derivation algorithm alg (PSA_ALG_XXX value such that PSA_ALG_IS_KEY_AGREEMENT(alg) is true and PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) is false). The operation must be ready for an input of the type given by step.

step — Which step the input data is for.

private_key — Identifier of the private key to use. It must allow the usage PSA_KEY_USAGE_DERIVE.

peer_key — Public key of the peer. The peer key must be in the same format that psa_import_key() accepts for the public key type corresponding to the type of private_key. That is, this function performs the equivalent of psa_import_key(..., peer_key, peer_key_length) where with key attributes indicating the public key type corresponding to the type of private_key. For example, for EC keys, this means that peer_key is

	interpreted as a point on the curve that the private key is on. The standard formats for public keys are documented in the documentation of `psa_export_public_key()`.
`peer_key_length`	Size of `peer_key` in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The operation state is not valid for this key agreement `step`.
`PSA_ERROR_INVALID_HANDLE`	
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`private_key` is not compatible with `alg`, or `peer_key` is not valid for `alg` or not compatible with `private_key`.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a key derivation algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	`step` does not allow an input resulting from a key agreement.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_HARDWARE_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	
`PSA_ERROR_BAD_STATE`	The library has not been previously initialized by `psa_crypto_init()`. It is implementation-dependent whether a failure to initialize results in this error code.

Description

A key agreement algorithm takes two inputs: a private key `private_key` a public key `peer_key`. The result of this function is passed as input to a key derivation. The output of this key derivation can be extracted by reading from the resulting operation to produce keys and other cryptographic material.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_key_derivation_abort()`.

10.9.4 Support macros

PSA_ALG_KEY_AGREEMENT_GET_BASE (macro)

Get the raw key agreement algorithm from a full key agreement algorithm.

```
#define PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) /* specification-defined value */
```

Parameters

`alg`	A key agreement algorithm identifier (value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_AGREEMENT(alg)` is true).

Returns

The underlying raw key agreement algorithm if `alg` is a key agreement algorithm.

Unspecified if `alg` is not a key agreement algorithm or if it is not supported by the implementation.

Description

See also `PSA_ALG_KEY_AGREEMENT()` and `PSA_ALG_KEY_AGREEMENT_GET_KDF()`.

PSA_ALG_KEY_AGREEMENT_GET_KDF (macro)

Get the key derivation algorithm used in a full key agreement algorithm.

```
#define PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) /* specification-defined value */
```

Parameters

`alg`	A key agreement algorithm identifier (value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_AGREEMENT(alg)` is true).

Returns

The underlying key derivation algorithm if `alg` is a key agreement algorithm.

Unspecified if `alg` is not a key agreement algorithm or if it is not supported by the implementation.

Description

See also `PSA_ALG_KEY_AGREEMENT()` and `PSA_ALG_KEY_AGREEMENT_GET_BASE()`.

PSA_ALG_IS_RAW_KEY_AGREEMENT (macro)

Whether the specified algorithm is a raw key agreement algorithm.

```
#define PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a raw key agreement algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

Description

A raw key agreement algorithm is one that does not specify a key derivation function. Usually, raw key agreement algorithms are constructed directly with a `PSA_ALG_xxx` macro while non-raw key agreement algorithms are constructed with `PSA_ALG_KEY_AGREEMENT()`.

The raw key agreement algorithm can be extracted from a full key agreement algorithm identifier using `PSA_ALG_KEY_AGREEMENT_GET_BASE()`.

PSA_ALG_IS_FFDH (macro)

Whether the specified algorithm is a finite field Diffie-Hellman algorithm.

```
#define PSA_ALG_IS_FFDH(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is a finite field Diffie-Hellman algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported key agreement algorithm identifier.

Description

This includes the raw finite field Diffie-Hellman algorithm as well as finite-field Diffie-Hellman followed by any supporter key derivation algorithm.

PSA_ALG_IS_ECDH (macro)

Whether the specified algorithm is an elliptic curve Diffie-Hellman algorithm.

```
#define PSA_ALG_IS_ECDH(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier (value of type `psa_algorithm_t`).

Returns

`1` if `alg` is an elliptic curve Diffie-Hellman algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported key agreement algorithm identifier.

Description

This includes the raw elliptic curve Diffie-Hellman algorithm as well as elliptic curve Diffie-Hellman followed by any supporter key derivation algorithm.

PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE (macro)

Sufficient output buffer size for `psa_raw_key_agreement()`.

```
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
```

Parameters

`key_type`	A supported key type.
`key_bits`	The size of the key in bits.

Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that `psa_raw_key_agreement()` will not fail with `PSA_ERROR_BUFFER_TOO_SMALL`. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or `0`. If the parameters are not valid, the return value is unspecified.

Description

This macro returns a compile-time constant if its arguments are compile-time constants.

> **Warning:** This function might evaluate its arguments multiple times or zero times. Providing arguments that have side effects will result in implementation-specific behavior, and is non-portable.

See also PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE.

PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE (macro)

Maximum size of the output from psa_raw_key_agreement().

```
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
```

This macro must expand to a compile-time constant integer. It is recommended that this value is the maximum size of the output any raw key agreement algorithm supported by the implementation, in bytes. The value must not be smaller than this maximum.

See also PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE().

10.10 Other cryptographic services

10.10.1 Random number generation

psa_generate_random (function)

Generate random bytes.

```
psa_status_t psa_generate_random(uint8_t * output,
                                 size_t output_size);
```

Parameters

output	Output buffer for the generated data.
output_size	Number of bytes to generate and output.

Returns: psa_status_t

PSA_SUCCESS

PSA_ERROR_NOT_SUPPORTED

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_HARDWARE_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_BAD_STATE	The library has not been previously initialized by psa_crypto_init(). It is implementation-dependent whether a failure to initialize results in this error code.

Description

Warning: This function **can** fail! Callers MUST check the return status and MUST NOT use the content of the output buffer if the return status is not PSA_SUCCESS.

Note:

To generate a key, use psa_generate_key() instead.

Appendix A: Example header file

Each implementation of the PSA Crypto API must provide a header file named `psa/crypto.h`, in which the API elements in this specification are defined.

This appendix provides a example of the `psa/crypto.h` header file with all of the API elements. This can be used as a starting point or reference for an implementation.

A.1 psa/crypto.h

```
typedef /* implementation-defined type */ psa_aead_operation_t;
typedef uint32_t psa_algorithm_t;
typedef /* implementation-defined type */ psa_cipher_operation_t;
typedef uint8_t psa_dh_family_t;
typedef uint8_t psa_ecc_family_t;
typedef /* implementation-defined type */ psa_hash_operation_t;
typedef /* implementation-defined type */ psa_key_attributes_t;
typedef /* implementation-defined type */ psa_key_derivation_operation_t;
typedef uint16_t psa_key_derivation_step_t;
typedef uint32_t psa_key_id_t;
typedef uint32_t psa_key_lifetime_t;
typedef uint32_t psa_key_location_t;
typedef uint8_t psa_key_persistence_t;
typedef uint16_t psa_key_type_t;
typedef uint32_t psa_key_usage_t;
typedef /* implementation-defined type */ psa_mac_operation_t;
typedef int32_t psa_status_t;
#define PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length) \
    /* implementation-defined value */
#define PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg, ciphertext_length) \
    /* implementation-defined value */
#define PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length) \
    /* implementation-defined value */
#define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length) \
    /* implementation-defined value */
#define PSA_AEAD_FINISH_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_AEAD_FINISH_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
#define PSA_AEAD_NONCE_LENGTH(key_type, alg) /* implementation-defined value */
#define PSA_AEAD_NONCE_MAX_SIZE /* implementation-defined value */
#define PSA_AEAD_OPERATION_INIT /* implementation-defined value */
#define PSA_AEAD_TAG_LENGTH(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_AEAD_TAG_MAX_SIZE /* implementation-defined value */
#define PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
#define PSA_AEAD_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_AEAD_VERIFY_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
#define PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG(aead_alg) \
```

(continues on next page)

(continued from previous page)

```
    /* specification-defined value */
#define PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, tag_length) \
    /* specification-defined value */
#define PSA_ALG_ANY_HASH ((psa_algorithm_t)0x020000ff)
#define PSA_ALG_CBC_MAC ((psa_algorithm_t)0x03c00100)
#define PSA_ALG_CBC_NO_PADDING ((psa_algorithm_t)0x04404000)
#define PSA_ALG_CBC_PKCS7 ((psa_algorithm_t)0x04404100)
#define PSA_ALG_CCM ((psa_algorithm_t)0x05500100)
#define PSA_ALG_CFB ((psa_algorithm_t)0x04c01100)
#define PSA_ALG_CHACHA20_POLY1305 ((psa_algorithm_t)0x05100500)
#define PSA_ALG_CMAC ((psa_algorithm_t)0x03c00200)
#define PSA_ALG_CTR ((psa_algorithm_t)0x04c01000)
#define PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) /* specification-defined value */
#define PSA_ALG_ECB_NO_PADDING ((psa_algorithm_t)0x04404400)
#define PSA_ALG_ECDH ((psa_algorithm_t)0x09020000)
#define PSA_ALG_ECDSA(hash_alg) /* specification-defined value */
#define PSA_ALG_ECDSA_ANY ((psa_algorithm_t) 0x06000600)
#define PSA_ALG_FFDH ((psa_algorithm_t)0x09010000)
#define PSA_ALG_FULL_LENGTH_MAC(mac_alg) /* specification-defined value */
#define PSA_ALG_GCM ((psa_algorithm_t)0x05500200)
#define PSA_ALG_GET_HASH(alg) /* specification-defined value */
#define PSA_ALG_HKDF(hash_alg) /* specification-defined value */
#define PSA_ALG_HMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_IS_AEAD(alg) /* specification-defined value */
#define PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER(alg) /* specification-defined value */
#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) /* specification-defined value */
#define PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) /* specification-defined value */
#define PSA_ALG_IS_CIPHER(alg) /* specification-defined value */
#define PSA_ALG_IS_DETERMINISTIC_ECDSA(alg) /* specification-defined value */
#define PSA_ALG_IS_ECDH(alg) /* specification-defined value */
#define PSA_ALG_IS_ECDSA(alg) /* specification-defined value */
#define PSA_ALG_IS_FFDH(alg) /* specification-defined value */
#define PSA_ALG_IS_HASH(alg) /* specification-defined value */
#define PSA_ALG_IS_HASH_AND_SIGN(alg) /* specification-defined value */
#define PSA_ALG_IS_HKDF(alg) /* specification-defined value */
#define PSA_ALG_IS_HMAC(alg) /* specification-defined value */
#define PSA_ALG_IS_KEY_AGREEMENT(alg) /* specification-defined value */
#define PSA_ALG_IS_KEY_DERIVATION(alg) /* specification-defined value */
#define PSA_ALG_IS_MAC(alg) /* specification-defined value */
#define PSA_ALG_IS_RANDOMIZED_ECDSA(alg) /* specification-defined value */
#define PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) /* specification-defined value */
#define PSA_ALG_IS_RSA_OAEP(alg) /* specification-defined value */
#define PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) /* specification-defined value */
#define PSA_ALG_IS_RSA_PSS(alg) /* specification-defined value */
#define PSA_ALG_IS_SIGN(alg) /* specification-defined value */
#define PSA_ALG_IS_SIGN_HASH(alg) /* specification-defined value */
#define PSA_ALG_IS_SIGN_MESSAGE(alg) /* specification-defined value */
#define PSA_ALG_IS_STREAM_CIPHER(alg) /* specification-defined value */
#define PSA_ALG_IS_TLS12_PRF(alg) /* specification-defined value */
#define PSA_ALG_IS_TLS12_PSK_TO_MS(alg) /* specification-defined value */
#define PSA_ALG_IS_WILDCARD(alg) /* specification-defined value */
#define PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg) \
```

(continues on next page)

(continued from previous page)

```
    /* specification-defined value */
#define PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) /* specification-defined value */
#define PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) /* specification-defined value */
#define PSA_ALG_MD2 ((psa_algorithm_t)0x02000001)
#define PSA_ALG_MD4 ((psa_algorithm_t)0x02000002)
#define PSA_ALG_MD5 ((psa_algorithm_t)0x02000003)
#define PSA_ALG_NONE ((psa_algorithm_t)0)
#define PSA_ALG_OFB ((psa_algorithm_t)0x04c01200)
#define PSA_ALG_RIPEMD160 ((psa_algorithm_t)0x02000004)
#define PSA_ALG_RSA_OAEP(hash_alg) /* specification-defined value */
#define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x07000200)
#define PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg) /* specification-defined value */
#define PSA_ALG_RSA_PKCS1V15_SIGN_RAW ((psa_algorithm_t) 0x06000200)
#define PSA_ALG_RSA_PSS(hash_alg) /* specification-defined value */
#define PSA_ALG_SHA3_224 ((psa_algorithm_t)0x02000010)
#define PSA_ALG_SHA3_256 ((psa_algorithm_t)0x02000011)
#define PSA_ALG_SHA3_384 ((psa_algorithm_t)0x02000012)
#define PSA_ALG_SHA3_512 ((psa_algorithm_t)0x02000013)
#define PSA_ALG_SHA_1 ((psa_algorithm_t)0x02000005)
#define PSA_ALG_SHA_224 ((psa_algorithm_t)0x02000008)
#define PSA_ALG_SHA_256 ((psa_algorithm_t)0x02000009)
#define PSA_ALG_SHA_384 ((psa_algorithm_t)0x0200000a)
#define PSA_ALG_SHA_512 ((psa_algorithm_t)0x0200000b)
#define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0200000c)
#define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0200000d)
#define PSA_ALG_SM3 ((psa_algorithm_t)0x02000014)
#define PSA_ALG_STREAM_CIPHER ((psa_algorithm_t)0x04800100)
#define PSA_ALG_TLS12_PRF(hash_alg) /* specification-defined value */
#define PSA_ALG_TLS12_PSK_TO_MS(hash_alg) /* specification-defined value */
#define PSA_ALG_TRUNCATED_MAC(mac_alg, mac_length) \
    /* specification-defined value */
#define PSA_ALG_XTS ((psa_algorithm_t)0x0440ff00)
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_BLOCK_CIPHER_BLOCK_LENGTH(type) /* specification-defined value */
#define PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE /* implementation-defined value */
#define PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_ENCRYPT_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_CIPHER_FINISH_OUTPUT_SIZE(key_type, alg) \
```

(continues on next page)

(continued from previous page)

```
    /* implementation-defined value */
#define PSA_CIPHER_IV_LENGTH(key_type, alg) /* implementation-defined value */
#define PSA_CIPHER_IV_MAX_SIZE /* implementation-defined value */
#define PSA_CIPHER_OPERATION_INIT /* implementation-defined value */
#define PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_CRYPTO_API_VERSION_MAJOR 1
#define PSA_CRYPTO_API_VERSION_MINOR 0
#define PSA_DH_FAMILY_RFC7919 ((psa_dh_family_t) 0x03)
#define PSA_ECC_FAMILY_BRAINPOOL_P_R1 ((psa_ecc_family_t) 0x30)
#define PSA_ECC_FAMILY_FRP ((psa_ecc_family_t) 0x33)
#define PSA_ECC_FAMILY_MONTGOMERY ((psa_ecc_family_t) 0x41)
#define PSA_ECC_FAMILY_SECP_K1 ((psa_ecc_family_t) 0x17)
#define PSA_ECC_FAMILY_SECP_R1 ((psa_ecc_family_t) 0x12)
#define PSA_ECC_FAMILY_SECP_R2 ((psa_ecc_family_t) 0x1b)
#define PSA_ECC_FAMILY_SECT_K1 ((psa_ecc_family_t) 0x27)
#define PSA_ECC_FAMILY_SECT_R1 ((psa_ecc_family_t) 0x22)
#define PSA_ECC_FAMILY_SECT_R2 ((psa_ecc_family_t) 0x2b)
#define PSA_ERROR_ALREADY_EXISTS ((psa_status_t)-139)
#define PSA_ERROR_BAD_STATE ((psa_status_t)-137)
#define PSA_ERROR_BUFFER_TOO_SMALL ((psa_status_t)-138)
#define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)-145)
#define PSA_ERROR_CORRUPTION_DETECTED ((psa_status_t)-151)
#define PSA_ERROR_DATA_CORRUPT ((psa_status_t)-152)
#define PSA_ERROR_DATA_INVALID ((psa_status_t)-153)
#define PSA_ERROR_DOES_NOT_EXIST ((psa_status_t)-140)
#define PSA_ERROR_GENERIC_ERROR ((psa_status_t)-132)
#define PSA_ERROR_HARDWARE_FAILURE ((psa_status_t)-147)
#define PSA_ERROR_INSUFFICIENT_DATA ((psa_status_t)-143)
#define PSA_ERROR_INSUFFICIENT_ENTROPY ((psa_status_t)-148)
#define PSA_ERROR_INSUFFICIENT_MEMORY ((psa_status_t)-141)
#define PSA_ERROR_INSUFFICIENT_STORAGE ((psa_status_t)-142)
#define PSA_ERROR_INVALID_ARGUMENT ((psa_status_t)-135)
#define PSA_ERROR_INVALID_HANDLE ((psa_status_t)-136)
#define PSA_ERROR_INVALID_PADDING ((psa_status_t)-150)
#define PSA_ERROR_INVALID_SIGNATURE ((psa_status_t)-149)
#define PSA_ERROR_NOT_PERMITTED ((psa_status_t)-133)
#define PSA_ERROR_NOT_SUPPORTED ((psa_status_t)-134)
#define PSA_ERROR_STORAGE_FAILURE ((psa_status_t)-146)
#define PSA_EXPORT_KEY_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
#define PSA_EXPORT_KEY_PAIR_MAX_SIZE /* implementation-defined value */
#define PSA_EXPORT_PUBLIC_KEY_MAX_SIZE /* implementation-defined value */
#define PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
#define PSA_HASH_BLOCK_LENGTH(alg) /* implementation-defined value */
#define PSA_HASH_LENGTH(alg) /* implementation-defined value */
#define PSA_HASH_MAX_SIZE /* implementation-defined value */
#define PSA_HASH_OPERATION_INIT /* implementation-defined value */
#define PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH ((size_t)4)
```

(continues on next page)

(continued from previous page)

```
#define PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) \
    /* specification-defined value */
#define PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) \
    /* specification-defined value */
#define PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) /* specification-defined value */
#define PSA_KEY_ATTRIBUTES_INIT /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_CONTEXT /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_INFO /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_LABEL /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_SALT /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_SECRET /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_SEED /* implementation-defined value */
#define PSA_KEY_DERIVATION_OPERATION_INIT /* implementation-defined value */
#define PSA_KEY_DERIVATION_UNLIMITED_CAPACITY \
    /* implementation-defined value */
#define PSA_KEY_ID_NULL ((psa_key_id_t)0)
#define PSA_KEY_ID_USER_MAX ((psa_key_id_t)0x3fffffff)
#define PSA_KEY_ID_USER_MIN ((psa_key_id_t)0x00000001)
#define PSA_KEY_ID_VENDOR_MAX ((psa_key_id_t)0x7fffffff)
#define PSA_KEY_ID_VENDOR_MIN ((psa_key_id_t)0x40000000)
#define PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(persistence, location) \
    ((location) << 8 | (persistence))
#define PSA_KEY_LIFETIME_GET_LOCATION(lifetime) \
    ((psa_key_location_t) ((lifetime) >> 8))
#define PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) \
    ((psa_key_persistence_t) ((lifetime) & 0x000000ff))
#define PSA_KEY_LIFETIME_IS_VOLATILE(lifetime) \
    (PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) == PSA_KEY_PERSISTENCE_VOLATILE)
#define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t) 0x00000001)
#define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t) 0x00000000)
#define PSA_KEY_LOCATION_LOCAL_STORAGE ((psa_key_location_t) 0x000000)
#define PSA_KEY_LOCATION_PRIMARY_SECURE_ELEMENT ((psa_key_location_t) 0x000001)
#define PSA_KEY_PERSISTENCE_DEFAULT ((psa_key_persistence_t) 0x01)
#define PSA_KEY_PERSISTENCE_READ_ONLY ((psa_key_persistence_t) 0xff)
#define PSA_KEY_PERSISTENCE_VOLATILE ((psa_key_persistence_t) 0x00)
#define PSA_KEY_TYPE_AES ((psa_key_type_t)0x2400)
#define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x2002)
#define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x2403)
#define PSA_KEY_TYPE_CHACHA20 ((psa_key_type_t)0x2004)
#define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x1200)
#define PSA_KEY_TYPE_DES ((psa_key_type_t)0x2301)
#define PSA_KEY_TYPE_DH_GET_FAMILY(type) /* specification-defined value */
#define PSA_KEY_TYPE_DH_KEY_PAIR(group) /* specification-defined value */
#define PSA_KEY_TYPE_DH_PUBLIC_KEY(group) /* specification-defined value */
#define PSA_KEY_TYPE_ECC_GET_FAMILY(type) /* specification-defined value */
#define PSA_KEY_TYPE_ECC_KEY_PAIR(curve) /* specification-defined value */
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) /* specification-defined value */
#define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x1100)
#define PSA_KEY_TYPE_IS_ASYMMETRIC(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_DH(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_DH_KEY_PAIR(type) /* specification-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_KEY_TYPE_IS_DH_PUBLIC_KEY(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_ECC(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_ECC_KEY_PAIR(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_KEY_PAIR(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_RSA(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_UNSTRUCTURED(type) /* specification-defined value */
#define PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x0000)
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x1001)
#define PSA_KEY_TYPE_RSA_KEY_PAIR ((psa_key_type_t)0x7001)
#define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x4001)
#define PSA_KEY_TYPE_SM4 ((psa_key_type_t)0x2405)
#define PSA_KEY_USAGE_CACHE ((psa_key_usage_t)0x00000004)
#define PSA_KEY_USAGE_COPY ((psa_key_usage_t)0x00000002)
#define PSA_KEY_USAGE_DECRYPT ((psa_key_usage_t)0x00000200)
#define PSA_KEY_USAGE_DERIVE ((psa_key_usage_t)0x00004000)
#define PSA_KEY_USAGE_ENCRYPT ((psa_key_usage_t)0x00000100)
#define PSA_KEY_USAGE_EXPORT ((psa_key_usage_t)0x00000001)
#define PSA_KEY_USAGE_SIGN_HASH ((psa_key_usage_t)0x00001000)
#define PSA_KEY_USAGE_SIGN_MESSAGE ((psa_key_usage_t)0x00000400)
#define PSA_KEY_USAGE_VERIFY_HASH ((psa_key_usage_t)0x00002000)
#define PSA_KEY_USAGE_VERIFY_MESSAGE ((psa_key_usage_t)0x00000800)
#define PSA_MAC_LENGTH(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_MAC_MAX_SIZE /* implementation-defined value */
#define PSA_MAC_OPERATION_INIT /* implementation-defined value */
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
#define PSA_SIGNATURE_MAX_SIZE /* implementation-defined value */
#define PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_SUCCESS ((psa_status_t)0)
#define PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE /* implementation-defined value */
psa_status_t psa_aead_abort(psa_aead_operation_t * operation);
psa_status_t psa_aead_decrypt(psa_key_id_t key,
                              psa_algorithm_t alg,
                              const uint8_t * nonce,
                              size_t nonce_length,
                              const uint8_t * additional_data,
                              size_t additional_data_length,
                              const uint8_t * ciphertext,
                              size_t ciphertext_length,
                              uint8_t * plaintext,
                              size_t plaintext_size,
                              size_t * plaintext_length);
```

(continues on next page)

(continued from previous page)

```
psa_status_t psa_aead_decrypt_setup(psa_aead_operation_t * operation,
                                    psa_key_id_t key,
                                    psa_algorithm_t alg);
psa_status_t psa_aead_encrypt(psa_key_id_t key,
                              psa_algorithm_t alg,
                              const uint8_t * nonce,
                              size_t nonce_length,
                              const uint8_t * additional_data,
                              size_t additional_data_length,
                              const uint8_t * plaintext,
                              size_t plaintext_length,
                              uint8_t * ciphertext,
                              size_t ciphertext_size,
                              size_t * ciphertext_length);
psa_status_t psa_aead_encrypt_setup(psa_aead_operation_t * operation,
                                    psa_key_id_t key,
                                    psa_algorithm_t alg);
psa_status_t psa_aead_finish(psa_aead_operation_t * operation,
                             uint8_t * ciphertext,
                             size_t ciphertext_size,
                             size_t * ciphertext_length,
                             uint8_t * tag,
                             size_t tag_size,
                             size_t * tag_length);
psa_status_t psa_aead_generate_nonce(psa_aead_operation_t * operation,
                                     uint8_t * nonce,
                                     size_t nonce_size,
                                     size_t * nonce_length);
psa_aead_operation_t psa_aead_operation_init(void);
psa_status_t psa_aead_set_lengths(psa_aead_operation_t * operation,
                                  size_t ad_length,
                                  size_t plaintext_length);
psa_status_t psa_aead_set_nonce(psa_aead_operation_t * operation,
                                const uint8_t * nonce,
                                size_t nonce_length);
psa_status_t psa_aead_update(psa_aead_operation_t * operation,
                             const uint8_t * input,
                             size_t input_length,
                             uint8_t * output,
                             size_t output_size,
                             size_t * output_length);
psa_status_t psa_aead_update_ad(psa_aead_operation_t * operation,
                                const uint8_t * input,
                                size_t input_length);
psa_status_t psa_aead_verify(psa_aead_operation_t * operation,
                             uint8_t * plaintext,
                             size_t plaintext_size,
                             size_t * plaintext_length,
                             const uint8_t * tag,
                             size_t tag_length);
psa_status_t psa_asymmetric_decrypt(psa_key_id_t key,
                                    psa_algorithm_t alg,
```

(continues on next page)

(continued from previous page)

```
                                    const uint8_t * input,
                                    size_t input_length,
                                    const uint8_t * salt,
                                    size_t salt_length,
                                    uint8_t * output,
                                    size_t output_size,
                                    size_t * output_length);
psa_status_t psa_asymmetric_encrypt(psa_key_id_t key,
                                    psa_algorithm_t alg,
                                    const uint8_t * input,
                                    size_t input_length,
                                    const uint8_t * salt,
                                    size_t salt_length,
                                    uint8_t * output,
                                    size_t output_size,
                                    size_t * output_length);
psa_status_t psa_cipher_abort(psa_cipher_operation_t * operation);
psa_status_t psa_cipher_decrypt(psa_key_id_t key,
                                psa_algorithm_t alg,
                                const uint8_t * input,
                                size_t input_length,
                                uint8_t * output,
                                size_t output_size,
                                size_t * output_length);
psa_status_t psa_cipher_decrypt_setup(psa_cipher_operation_t * operation,
                                      psa_key_id_t key,
                                      psa_algorithm_t alg);
psa_status_t psa_cipher_encrypt(psa_key_id_t key,
                                psa_algorithm_t alg,
                                const uint8_t * input,
                                size_t input_length,
                                uint8_t * output,
                                size_t output_size,
                                size_t * output_length);
psa_status_t psa_cipher_encrypt_setup(psa_cipher_operation_t * operation,
                                      psa_key_id_t key,
                                      psa_algorithm_t alg);
psa_status_t psa_cipher_finish(psa_cipher_operation_t * operation,
                               uint8_t * output,
                               size_t output_size,
                               size_t * output_length);
psa_status_t psa_cipher_generate_iv(psa_cipher_operation_t * operation,
                                    uint8_t * iv,
                                    size_t iv_size,
                                    size_t * iv_length);
psa_cipher_operation_t psa_cipher_operation_init(void);
psa_status_t psa_cipher_set_iv(psa_cipher_operation_t * operation,
                               const uint8_t * iv,
                               size_t iv_length);
psa_status_t psa_cipher_update(psa_cipher_operation_t * operation,
                               const uint8_t * input,
                               size_t input_length,
```

(continues on next page)

(continued from previous page)

```
                               uint8_t * output,
                               size_t output_size,
                               size_t * output_length);
psa_status_t psa_copy_key(psa_key_id_t source_key,
                          const psa_key_attributes_t * attributes,
                          psa_key_id_t * target_key);
psa_status_t psa_crypto_init(void);
psa_status_t psa_destroy_key(psa_key_id_t key);
psa_status_t psa_export_key(psa_key_id_t key,
                            uint8_t * data,
                            size_t data_size,
                            size_t * data_length);
psa_status_t psa_export_public_key(psa_key_id_t key,
                                   uint8_t * data,
                                   size_t data_size,
                                   size_t * data_length);
psa_status_t psa_generate_key(const psa_key_attributes_t * attributes,
                              psa_key_id_t * key);
psa_status_t psa_generate_random(uint8_t * output,
                                 size_t output_size);
psa_algorithm_t psa_get_key_algorithm(const psa_key_attributes_t * attributes);
psa_status_t psa_get_key_attributes(psa_key_id_t key,
                                    psa_key_attributes_t * attributes);
size_t psa_get_key_bits(const psa_key_attributes_t * attributes);
psa_key_id_t psa_get_key_id(const psa_key_attributes_t * attributes);
psa_key_lifetime_t psa_get_key_lifetime(const psa_key_attributes_t * attributes);
psa_key_type_t psa_get_key_type(const psa_key_attributes_t * attributes);
psa_key_usage_t psa_get_key_usage_flags(const psa_key_attributes_t * attributes);
psa_status_t psa_hash_abort(psa_hash_operation_t * operation);
psa_status_t psa_hash_clone(const psa_hash_operation_t * source_operation,
                            psa_hash_operation_t * target_operation);
psa_status_t psa_hash_compare(psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
                              const uint8_t * hash,
                              size_t hash_length);
psa_status_t psa_hash_compute(psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
                              uint8_t * hash,
                              size_t hash_size,
                              size_t * hash_length);
psa_status_t psa_hash_finish(psa_hash_operation_t * operation,
                             uint8_t * hash,
                             size_t hash_size,
                             size_t * hash_length);
psa_hash_operation_t psa_hash_operation_init(void);
psa_status_t psa_hash_resume(psa_hash_operation_t * operation,
                             const uint8_t * hash_state,
                             size_t hash_state_length);
psa_status_t psa_hash_setup(psa_hash_operation_t * operation,
                            psa_algorithm_t alg);
```

(continues on next page)

(continued from previous page)

```
psa_status_t psa_hash_suspend(psa_hash_operation_t * operation,
                              uint8_t * hash_state,
                              size_t hash_state_size,
                              size_t * hash_state_length);
psa_status_t psa_hash_update(psa_hash_operation_t * operation,
                             const uint8_t * input,
                             size_t input_length);
psa_status_t psa_hash_verify(psa_hash_operation_t * operation,
                             const uint8_t * hash,
                             size_t hash_length);
psa_status_t psa_import_key(const psa_key_attributes_t * attributes,
                            const uint8_t * data,
                            size_t data_length,
                            psa_key_id_t * key);
psa_key_attributes_t psa_key_attributes_init(void);
psa_status_t psa_key_derivation_abort(psa_key_derivation_operation_t * operation);
psa_status_t psa_key_derivation_get_capacity(const psa_key_derivation_operation_t * operation,
                                             size_t * capacity);
psa_status_t psa_key_derivation_input_bytes(psa_key_derivation_operation_t * operation,
                                            psa_key_derivation_step_t step,
                                            const uint8_t * data,
                                            size_t data_length);
psa_status_t psa_key_derivation_input_key(psa_key_derivation_operation_t * operation,
                                          psa_key_derivation_step_t step,
                                          psa_key_id_t key);
psa_status_t psa_key_derivation_key_agreement(psa_key_derivation_operation_t * operation,
                                              psa_key_derivation_step_t step,
                                              psa_key_id_t private_key,
                                              const uint8_t * peer_key,
                                              size_t peer_key_length);
psa_key_derivation_operation_t psa_key_derivation_operation_init(void);
psa_status_t psa_key_derivation_output_bytes(psa_key_derivation_operation_t * operation,
                                             uint8_t * output,
                                             size_t output_length);
psa_status_t psa_key_derivation_output_key(const psa_key_attributes_t * attributes,
                                           psa_key_derivation_operation_t * operation,
                                           psa_key_id_t * key);
psa_status_t psa_key_derivation_set_capacity(psa_key_derivation_operation_t * operation,
                                             size_t capacity);
psa_status_t psa_key_derivation_setup(psa_key_derivation_operation_t * operation,
                                      psa_algorithm_t alg);
psa_status_t psa_mac_abort(psa_mac_operation_t * operation);
psa_status_t psa_mac_compute(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const uint8_t * input,
                             size_t input_length,
                             uint8_t * mac,
                             size_t mac_size,
                             size_t * mac_length);
psa_mac_operation_t psa_mac_operation_init(void);
psa_status_t psa_mac_sign_finish(psa_mac_operation_t * operation,
                                 uint8_t * mac,
```

(continues on next page)

(continued from previous page)

```
                                    size_t mac_size,
                                    size_t * mac_length);
psa_status_t psa_mac_sign_setup(psa_mac_operation_t * operation,
                                psa_key_id_t key,
                                psa_algorithm_t alg);
psa_status_t psa_mac_update(psa_mac_operation_t * operation,
                            const uint8_t * input,
                            size_t input_length);
psa_status_t psa_mac_verify(psa_key_id_t key,
                            psa_algorithm_t alg,
                            const uint8_t * input,
                            size_t input_length,
                            const uint8_t * mac,
                            size_t mac_length);
psa_status_t psa_mac_verify_finish(psa_mac_operation_t * operation,
                                   const uint8_t * mac,
                                   size_t mac_length);
psa_status_t psa_mac_verify_setup(psa_mac_operation_t * operation,
                                  psa_key_id_t key,
                                  psa_algorithm_t alg);
psa_status_t psa_purge_key(psa_key_id_t key);
psa_status_t psa_raw_key_agreement(psa_algorithm_t alg,
                                   psa_key_id_t private_key,
                                   const uint8_t * peer_key,
                                   size_t peer_key_length,
                                   uint8_t * output,
                                   size_t output_size,
                                   size_t * output_length);
void psa_reset_key_attributes(psa_key_attributes_t * attributes);
void psa_set_key_algorithm(psa_key_attributes_t * attributes,
                           psa_algorithm_t alg);
void psa_set_key_bits(psa_key_attributes_t * attributes,
                      size_t bits);
void psa_set_key_id(psa_key_attributes_t * attributes,
                    psa_key_id_t id);
void psa_set_key_lifetime(psa_key_attributes_t * attributes,
                          psa_key_lifetime_t lifetime);
void psa_set_key_type(psa_key_attributes_t * attributes,
                      psa_key_type_t type);
void psa_set_key_usage_flags(psa_key_attributes_t * attributes,
                             psa_key_usage_t usage_flags);
psa_status_t psa_sign_hash(psa_key_id_t key,
                           psa_algorithm_t alg,
                           const uint8_t * hash,
                           size_t hash_length,
                           uint8_t * signature,
                           size_t signature_size,
                           size_t * signature_length);
psa_status_t psa_sign_message(psa_key_id_t key,
                              psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
```

(continues on next page)

(continued from previous page)

```
                              uint8_t * signature,
                              size_t signature_size,
                              size_t * signature_length);
psa_status_t psa_verify_hash(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const uint8_t * hash,
                             size_t hash_length,
                             const uint8_t * signature,
                             size_t signature_length);
psa_status_t psa_verify_message(psa_key_id_t key,
                                psa_algorithm_t alg,
                                const uint8_t * input,
                                size_t input_length,
                                const uint8_t * signature,
                                size_t signature_length);
```

Appendix B: Example macro implementations

This appendix provides example implementations of the function-like macros that have specification-defined values.

> **Note:**
>
> In a future version of this specification, these example implementations will be replaced with a pseudo-code representation of the macro's computation in the macro description.

The examples here provide correct results for the valid inputs defined by each API, for an implementation that supports all of the defined algorithms and key types. An implementation can provide alternative definitions of these macros:

- If the implementation does not support all of the algorithms or key types, it can provide a simpler definition of applicable macros.
- If the implementation provides vendor-specific algorithms or key types, it needs to extend the definitions of applicable macros.

B.1 Algorithm macros

```
#define PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG(aead_alg) \
    ((((aead_alg) & ~0x003f0000) == 0x05400100) ? PSA_ALG_CCM : \
     (((aead_alg) & ~0x003f0000) == 0x05400200) ? PSA_ALG_GCM : \
     (((aead_alg) & ~0x003f0000) == 0x05000500) ? PSA_ALG_CHACHA20_POLY1305 : \
     PSA_ALG_NONE)

#define PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, tag_length) \
    ((psa_algorithm_t) (((aead_alg) & ~0x003f0000) | (((tag_length) & 0x3f) << 16)))
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) \
    ((psa_algorithm_t) (0x06000700 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_ECDSA(hash_alg) \
    ((psa_algorithm_t) (0x06000600 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_FULL_LENGTH_MAC(mac_alg) \
    ((psa_algorithm_t) ((mac_alg) & ~0x003f0000))

#define PSA_ALG_GET_HASH(alg) \
    (((alg) & 0x000000ff) == 0 ? PSA_ALG_NONE : 0x02000000 | ((alg) & 0x000000ff))

#define PSA_ALG_HKDF(hash_alg) \
    ((psa_algorithm_t) (0x08000100 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x03800000 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_IS_AEAD(alg) \
    (((alg) & 0x7f000000) == 0x05000000)

#define PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER(alg) \
    (((alg) & 0x7f400000) == 0x05400000)

#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) \
    (((alg) & 0x7f000000) == 0x07000000)

#define PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) \
    (((alg) & 0x7fc00000) == 0x03c00000)

#define PSA_ALG_IS_CIPHER(alg) \
    (((alg) & 0x7f000000) == 0x03000000)

#define PSA_ALG_IS_DETERMINISTIC_ECDSA(alg) \
    (((alg) & ~0x000000ff) == 0x06000700)

#define PSA_ALG_IS_ECDH(alg) \
    (((alg) & 0x7fff0000) == 0x09020000)

#define PSA_ALG_IS_ECDSA(alg) \
    (((alg) & ~0x000001ff) == 0x06000600)

#define PSA_ALG_IS_FFDH(alg) \
    (((alg) & 0x7fff0000) == 0x09010000)

#define PSA_ALG_IS_HASH(alg) \
    (((alg) & 0x7f000000) == 0x02000000)

#define PSA_ALG_IS_HASH_AND_SIGN(alg) \
    (PSA_ALG_IS_RSA_PSS(alg) || PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) || PSA_ALG_IS_ECDSA(alg))
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_IS_HKDF(alg) \
    (((alg) & ~0x000000ff) == 0x08000100)

#define PSA_ALG_IS_HMAC(alg) \
    (((alg) & 0x7fc0ff00) == 0x03800000)

#define PSA_ALG_IS_KEY_AGREEMENT(alg) \
    (((alg) & 0x7f000000) == 0x09000000)

#define PSA_ALG_IS_KEY_DERIVATION(alg) \
    (((alg) & 0x7f000000) == 0x08000000)

#define PSA_ALG_IS_MAC(alg) \
    (((alg) & 0x7f000000) == 0x03000000)

#define PSA_ALG_IS_RANDOMIZED_ECDSA(alg) \
    (((alg) & ~0x000000ff) == 0x06000600)

#define PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) \
    (((alg) & 0x7f00ffff) == 0x09000000)

#define PSA_ALG_IS_RSA_OAEP(alg) \
    (((alg) & ~0x000000ff) == 0x07000300)

#define PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) \
    (((alg) & ~0x000000ff) == 0x06000200)

#define PSA_ALG_IS_RSA_PSS(alg) \
    (((alg) & ~0x000000ff) == 0x06000300)

#define PSA_ALG_IS_SIGN(alg) \
    (((alg) & 0x7f000000) == 0x06000000)

#define PSA_ALG_IS_SIGN_HASH(alg) \
    PSA_ALG_IS_SIGN(alg)

#define PSA_ALG_IS_SIGN_MESSAGE(alg) \
    (PSA_ALG_IS_SIGN(alg) && \
     (alg) != PSA_ALG_ECDSA_ANY && (alg) != PSA_ALG_RSA_PKCS1V15_SIGN_RAW)

#define PSA_ALG_IS_STREAM_CIPHER(alg) \
    (((alg) & 0x7f800000) == 0x04800000)

#define PSA_ALG_IS_TLS12_PRF(alg) \
    (((alg) & ~0x000000ff) == 0x08000200)

#define PSA_ALG_IS_TLS12_PSK_TO_MS(alg) \
    (((alg) & ~0x000000ff) == 0x08000300)

#define PSA_ALG_IS_WILDCARD(alg) \
    (PSA_ALG_GET_HASH(alg) == PSA_ALG_HASH_ANY)
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg) \
    ((ka_alg) | (kdf_alg))

#define PSA_ALG_KEY_AGREEMENT_GET_BASE(alg) \
    ((psa_algorithm_t)((alg) & 0xffff0000))

#define PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) \
    ((psa_algorithm_t)((alg) & 0xfe00ffff))

#define PSA_ALG_RSA_OAEP(hash_alg) \
    ((psa_algorithm_t)(0x07000300 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg) \
    ((psa_algorithm_t)(0x06000200 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_RSA_PSS(hash_alg) \
    ((psa_algorithm_t)(0x06000300 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_TLS12_PRF(hash_alg) \
    ((psa_algorithm_t) (0x08000200 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_TLS12_PSK_TO_MS(hash_alg) \
    ((psa_algorithm_t) (0x08000300 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_TRUNCATED_MAC(mac_alg, mac_length) \
    ((psa_algorithm_t) (((mac_alg) & ~0x003f0000) | (((mac_length) & 0x3f) << 16)))
```

B.2 Key type macros

```
#define PSA_BLOCK_CIPHER_BLOCK_LENGTH(type) \
    (1u << (((type) >> 8) & 7))

#define PSA_KEY_TYPE_DH_GET_FAMILY(type) \
    ((psa_dh_family_t) ((type) & 0x00ff))

#define PSA_KEY_TYPE_DH_KEY_PAIR(group) \
    ((psa_key_type_t) (0x7200 | (group)))

#define PSA_KEY_TYPE_DH_PUBLIC_KEY(group) \
    ((psa_key_type_t) (0x4200 | (group)))

#define PSA_KEY_TYPE_ECC_GET_FAMILY(type) \
    ((psa_ecc_family_t) ((type) & 0x00ff))

#define PSA_KEY_TYPE_ECC_KEY_PAIR(curve) \
    ((psa_key_type_t) (0x7100 | (curve)))

#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) \
    ((psa_key_type_t) (0x4100 | (curve)))

#define PSA_KEY_TYPE_IS_ASYMMETRIC(type) \
```

(continues on next page)

(continued from previous page)

```
    (((type) & 0x4000) == 0x4000)

#define PSA_KEY_TYPE_IS_DH(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff00) == 0x4200)

#define PSA_KEY_TYPE_IS_DH_KEY_PAIR(type) \
    (((type) & 0xff00) == 0x7200)

#define PSA_KEY_TYPE_IS_DH_PUBLIC_KEY(type) \
    (((type) & 0xff00) == 0x4200)

#define PSA_KEY_TYPE_IS_ECC(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff00) == 0x4100)

#define PSA_KEY_TYPE_IS_ECC_KEY_PAIR(type) \
    (((type) & 0xff00) == 0x7100)

#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(type) \
    (((type) & 0xff00) == 0x4100)

#define PSA_KEY_TYPE_IS_KEY_PAIR(type) \
    (((type) & 0x7000) == 0x7000)

#define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) \
    (((type) & 0x7000) == 0x4000)

#define PSA_KEY_TYPE_IS_RSA(type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) == 0x4001)

#define PSA_KEY_TYPE_IS_UNSTRUCTURED(type) \
    (((type) & 0x7000) == 0x1000 || ((type) & 0x7000) == 0x2000)

#define PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type) \
    ((psa_key_type_t) ((type) | 0x3000))

#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) \
    ((psa_key_type_t) ((type) & ~0x3000))
```

B.3 Hash suspend state macros

```
#define PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) \
    ((alg)==PSA_ALG_MD2 ? 64 : \
     (alg)==PSA_ALG_MD4 || (alg)==PSA_ALG_MD5 ? 16 : \
     (alg)==PSA_ALG_RIPEMD160 || (alg)==PSA_ALG_SHA_1 ? 20 : \
     (alg)==PSA_ALG_SHA_224 || (alg)==PSA_ALG_SHA_256 ? 32 : \
     (alg)==PSA_ALG_SHA_512 || (alg)==PSA_ALG_SHA_384 || (alg)==PSA_ALG_SHA_512_256 ? 64 : \
     0)

#define PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) \
    ((alg)==PSA_ALG_MD2 ? 1 : \
     (alg)==PSA_ALG_MD4 || (alg)==PSA_ALG_MD5 || (alg)==PSA_ALG_RIPEMD160 || \
```

(continues on next page)

(continued from previous page)

```
    (alg)==PSA_ALG_SHA_1 || (alg)==PSA_ALG_SHA_224 || (alg)==PSA_ALG_SHA_256 ? 8 : \
    (alg)==PSA_ALG_SHA_512 || (alg)==PSA_ALG_SHA_384 || (alg)==PSA_ALG_SHA_512_256 ? 16 : \
    0)

#define PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) \
    (PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH + \
     PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) + \
     PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) + \
     PSA_HASH_BLOCK_LENGTH(alg) - 1)
```

Appendix C: Changes to the API

C.1 Document change history

This section provides the detailed changes made between published version of the document.

C.1.1 Changes between *1.0.0* and *1.0.1*

Changes to the API

- Added subtypes `psa_key_persistence_t` and `psa_key_location_t` for key lifetimes, and defined standard values for these attributes.
- Added identifiers for `PSA_ALG_SM3` and `PSA_KEY_TYPE_SM4`.

Clarifications and fixes

- Provided citation references for all cryptographic algorithms in the specification.
- Provided precise key size information for all key types.
- Permitted implementations to store and export long HMAC keys in hashed form.
- Provided details for initialization vectors in all unauthenticated cipher algorithms.
- Provided details for nonces in all AEAD algorithms.
- Clarified the input steps for HKDF.
- Provided details of signature algorithms, include requirements when using with `psa_sign_hash()` and `psa_verify_hash()`.
- Provided details of key agreement algorithms, and how to use them.
- Aligned terminology relating to key policies, to clarify the combination of the usage flags and permitted algorithm in the policy.
- Clarified the use of the individual key attributes for all of the key creation functions.
- Restructured the description for `psa_key_derivation_output_key()`, to clarify the handling of the excess bits in ECC key generation when needing a string of bits whose length is not a multiple of `8`.
- Referenced the correct buffer size macros for `psa_export_key()`.

- Removed the use of the PSA_ERROR_DOES_NOT_EXIST error.
- Clarified concurrency rules.
- Document that psa_key_derivation_output_key() does not return PSA_ERROR_NOT_PERMITTED if the secret input is the result of a key agreement. This matches what was already documented for PSA_KEY_DERIVATION_INPUT_SECRET.
- Relax the requirement to use the defined key derivation methods in psa_key_derivation_output_key(): implementation-specific KDF algorithms can use implementation-defined methods to derive the key material.

Other changes

- Provided a glossary of terms.
- Provided a table of references.
- Restructured the *Key management reference* on page 49 chapter.
  - Moved individual attribute types, values and accessor functions into their own sections.
  - Placed permitted algorithms and usage flags into *Key policies* on page 78.
  - Moved most introductory material from the *Functionality overview* on page 20 into the relevant API sections.

C.1.2 Changes between *1.0 beta 3* and *1.0.0*

Changes to the API

- Added PSA_CRYPTO_API_VERSION_MAJOR and PSA_CRYPTO_API_VERSION_MINOR to report the PSA Crypto API version.
- Removed PSA_ALG_GMAC algorithm identifier.
- Removed internal implementation macros from the API specification:
  - PSA_AEAD_TAG_LENGTH_OFFSET
  - PSA_ALG_AEAD_FROM_BLOCK_FLAG
  - PSA_ALG_AEAD_TAG_LENGTH_MASK
  - PSA__ALG_AEAD_WITH_DEFAULT_TAG_LENGTH__CASE
  - PSA_ALG_CATEGORY_AEAD
  - PSA_ALG_CATEGORY_ASYMMETRIC_ENCRYPTION
  - PSA_ALG_CATEGORY_CIPHER
  - PSA_ALG_CATEGORY_HASH
  - PSA_ALG_CATEGORY_KEY_AGREEMENT
  - PSA_ALG_CATEGORY_KEY_DERIVATION
  - PSA_ALG_CATEGORY_MAC
  - PSA_ALG_CATEGORY_MASK
  - PSA_ALG_CATEGORY_SIGN
  - PSA_ALG_CIPHER_FROM_BLOCK_FLAG
  - PSA_ALG_CIPHER_MAC_BASE
  - PSA_ALG_CIPHER_STREAM_FLAG

- PSA_ALG_DETERMINISTIC_ECDSA_BASE
- PSA_ALG_ECDSA_BASE
- PSA_ALG_ECDSA_IS_DETERMINISTIC
- PSA_ALG_HASH_MASK
- PSA_ALG_HKDF_BASE
- PSA_ALG_HMAC_BASE
- PSA_ALG_IS_KEY_DERIVATION_OR_AGREEMENT
- PSA_ALG_IS_VENDOR_DEFINED
- PSA_ALG_KEY_AGREEMENT_MASK
- PSA_ALG_KEY_DERIVATION_MASK
- PSA_ALG_MAC_SUBCATEGORY_MASK
- PSA_ALG_MAC_TRUNCATION_MASK
- PSA_ALG_RSA_OAEP_BASE
- PSA_ALG_RSA_PKCS1V15_SIGN_BASE
- PSA_ALG_RSA_PSS_BASE
- PSA_ALG_TLS12_PRF_BASE
- PSA_ALG_TLS12_PSK_TO_MS_BASE
- PSA_ALG_VENDOR_FLAG
- PSA_BITS_TO_BYTES
- PSA_BYTES_TO_BITS
- PSA_ECDSA_SIGNATURE_SIZE
- PSA_HMAC_MAX_HASH_BLOCK_SIZE
- PSA_KEY_EXPORT_ASN1_INTEGER_MAX_SIZE
- PSA_KEY_EXPORT_DSA_KEY_PAIR_MAX_SIZE
- PSA_KEY_EXPORT_DSA_PUBLIC_KEY_MAX_SIZE
- PSA_KEY_EXPORT_ECC_KEY_PAIR_MAX_SIZE
- PSA_KEY_EXPORT_ECC_PUBLIC_KEY_MAX_SIZE
- PSA_KEY_EXPORT_RSA_KEY_PAIR_MAX_SIZE
- PSA_KEY_EXPORT_RSA_PUBLIC_KEY_MAX_SIZE
- PSA_KEY_TYPE_CATEGORY_FLAG_PAIR
- PSA_KEY_TYPE_CATEGORY_KEY_PAIR
- PSA_KEY_TYPE_CATEGORY_MASK
- PSA_KEY_TYPE_CATEGORY_PUBLIC_KEY
- PSA_KEY_TYPE_CATEGORY_RAW
- PSA_KEY_TYPE_CATEGORY_SYMMETRIC
- PSA_KEY_TYPE_DH_GROUP_MASK
- PSA_KEY_TYPE_DH_KEY_PAIR_BASE
- PSA_KEY_TYPE_DH_PUBLIC_KEY_BASE
- PSA_KEY_TYPE_ECC_CURVE_MASK
- PSA_KEY_TYPE_ECC_KEY_PAIR_BASE
- PSA_KEY_TYPE_ECC_PUBLIC_KEY_BASE
- PSA_KEY_TYPE_IS_VENDOR_DEFINED

- PSA_KEY_TYPE_VENDOR_FLAG
- PSA_MAC_TRUNCATED_LENGTH
- PSA_MAC_TRUNCATION_OFFSET
- PSA_ROUND_UP_TO_MULTIPLE
- PSA_RSA_MINIMUM_PADDING_SIZE
- PSA_VENDOR_ECC_MAX_CURVE_BITS
- PSA_VENDOR_RSA_MAX_KEY_BITS

- Remove the definition of implementation-defined macros from the specification, and clarified the implementation requirements for these macros in *Implementation-specific macros* on page 36.
  - Macros with implementation-defined values are indicated by `/* implementation-defined value */` in the API prototype. The implementation must provide the implementation.
  - Macros for algorithm and key type construction and inspection have specification-defined values. This is indicated by `/* specification-defined value */` in the API prototype. Example definitions of these macros is provided in *Example macro implementations* on page 237.
- Changed the semantics of multi-part operations.
  - Formalize the standard pattern for multi-part operations.
  - Require all errors to result in an error state, requiring a call to `psa_xxx_abort()` to reset the object.
  - Define behavior in illegal and impossible operation states, and for copying and reusing operation objects.

  Although the API signatures have not changed, this change requires modifications to application flows that handle error conditions in multi-part operations.
- Merge the key identifier and key handle concepts in the API.
  - Replaced all references to key handles with key identifiers, or something similar.
  - Replaced all uses of `psa_key_handle_t` with `psa_key_id_t` in the API, and removes the `psa_key_handle_t` type.
  - Removed `psa_open_key` and `psa_close_key`.
  - Added `PSA_KEY_ID_NULL` for the never valid zero key identifier.
  - Document rules related to destroying keys whilst in use.
  - Added the `PSA_KEY_USAGE_CACHE` usage flag and the related `psa_purge_key()` API.
  - Added clarification about caching keys to non-volatile memory.
- Renamed `PSA_ALG_TLS12_PSK_TO_MS_MAX_PSK_LEN` to `PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE`.
- Relax definition of implementation-defined types.
  - This is indicated in the specification by `/* implementation-defined type */` in the type definition.
  - The specification only defines the name of implementation-defined types, and does not require that the implementation is a C struct.
- Zero-length keys are not permitted. Attempting to create one will now result in an error.
- Relax the constraints on inputs to key derivation:
  - `psa_key_derivation_input_bytes()` can be used for secret input steps. This is necessary if a zero-length input is required by the application.
  - `psa_key_derivation_input_key()` can be used for non-secret input steps.

- Multi-part cipher operations now require that the IV is passed using psa_cipher_set_iv(), the option to provide this as part of the input to psa_cipher_update() has been removed.

  The format of the output from psa_cipher_encrypt(), and input to psa_cipher_decrypt(), is documented.

- Support macros to calculate the size of output buffers, IVs and nonces.
  - Macros to calculate a key and/or algorithm specific result are provided for all output buffers. The new macros are:
    - PSA_AEAD_NONCE_LENGTH()
    - PSA_CIPHER_ENCRYPT_OUTPUT_SIZE()
    - PSA_CIPHER_DECRYPT_OUTPUT_SIZE()
    - PSA_CIPHER_UPDATE_OUTPUT_SIZE()
    - PSA_CIPHER_FINISH_OUTPUT_SIZE()
    - PSA_CIPHER_IV_LENGTH()
    - PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE()
    - PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE()
  - Macros that evaluate to a maximum type-independent buffer size are provided. The new macros are:
    - PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE()
    - PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE()
    - PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE()
    - PSA_AEAD_FINISH_OUTPUT_MAX_SIZE
    - PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE
    - PSA_AEAD_NONCE_MAX_SIZE
    - PSA_AEAD_TAG_MAX_SIZE
    - PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE
    - PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE
    - PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE()
    - PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE()
    - PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE()
    - PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE
    - PSA_CIPHER_IV_MAX_SIZE
    - PSA_EXPORT_KEY_PAIR_MAX_SIZE
    - PSA_EXPORT_PUBLIC_KEY_MAX_SIZE
    - PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE
  - AEAD output buffer size macros are now parameterized on the key type as well as the algorithm:
    - PSA_AEAD_ENCRYPT_OUTPUT_SIZE()
    - PSA_AEAD_DECRYPT_OUTPUT_SIZE()
    - PSA_AEAD_UPDATE_OUTPUT_SIZE()
    - PSA_AEAD_FINISH_OUTPUT_SIZE()
    - PSA_AEAD_TAG_LENGTH()
    - PSA_AEAD_VERIFY_OUTPUT_SIZE()

- Some existing macros have been renamed to ensure that the name of the support macros are consistent. The following macros have been renamed:
  - PSA_ALG_AEAD_WITH_DEFAULT_TAG_LENGTH() → PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG()
  - PSA_ALG_AEAD_WITH_TAG_LENGTH() → PSA_ALG_AEAD_WITH_SHORTENED_TAG()
  - PSA_KEY_EXPORT_MAX_SIZE() → PSA_EXPORT_KEY_OUTPUT_SIZE()
  - PSA_HASH_SIZE() → PSA_HASH_LENGTH()
  - PSA_MAC_FINAL_SIZE() → PSA_MAC_LENGTH()
  - PSA_BLOCK_CIPHER_BLOCK_SIZE() → PSA_BLOCK_CIPHER_BLOCK_LENGTH()
  - PSA_MAX_BLOCK_CIPHER_BLOCK_SIZE → PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE
- Documentation of the macros and of related APIs has been updated to reference the related API elements.

- Provide hash-and-sign operations as well as sign-the-hash operations. The API for asymmetric signature has been changed to clarify the use of the new functions.
  - The existing asymmetric signature API has been renamed to clarify that this is for signing a hash that is already computed:
    - PSA_KEY_USAGE_SIGN → PSA_KEY_USAGE_SIGN_HASH
    - PSA_KEY_USAGE_VERIFY → PSA_KEY_USAGE_VERIFY_HASH
    - psa_asymmetric_sign() → psa_sign_hash()
    - psa_asymmetric_verify() → psa_verify_hash()
  - New APIs added to provide the complete message signing operation:
    - PSA_KEY_USAGE_SIGN_MESSAGE
    - PSA_KEY_USAGE_VERIFY_MESSAGE
    - psa_sign_message()
    - psa_verify_message()
  - New Support macros to identify which algorithms can be used in which signing API:
    - PSA_ALG_IS_SIGN_HASH()
    - PSA_ALG_IS_SIGN_MESSAGE()
  - Renamed support macros that apply to both signing APIs:
    - PSA_ASYMMETRIC_SIGN_OUTPUT_SIZE() → PSA_SIGN_OUTPUT_SIZE()
    - PSA_ASYMMETRIC_SIGNATURE_MAX_SIZE → PSA_SIGNATURE_MAX_SIZE
  - The usage flag values have been changed, including for PSA_KEY_USAGE_DERIVE.

- Restructure psa_key_type_t and reassign all key type values.
  - psa_key_type_t changes from 32-bit to 16-bit integer.
  - Reassigned the key type categories.
  - Add a parity bit to the key type to ensure that valid key type values differ by at least 2 bits.
  - 16-bit elliptic curve ids (psa_ecc_curve_t) replaced by 8-bit ECC curve family ids (psa_ecc_family_t). 16-bit Diffie-Hellman group ids (psa_dh_group_t) replaced by 8-bit DH group family ids (psa_dh_family_t).
    - These ids are no longer related to the IANA Group Registry specification.
    - The new key type values do not encode the key size for ECC curves or DH groups. The key bit size from the key attributes identify a specific ECC curve or DH group within the family.
  - The following macros have been removed:

- PSA_DH_GROUP_FFDHE2048
- PSA_DH_GROUP_FFDHE3072
- PSA_DH_GROUP_FFDHE4096
- PSA_DH_GROUP_FFDHE6144
- PSA_DH_GROUP_FFDHE8192
- PSA_ECC_CURVE_BITS
- PSA_ECC_CURVE_BRAINPOOL_P256R1
- PSA_ECC_CURVE_BRAINPOOL_P384R1
- PSA_ECC_CURVE_BRAINPOOL_P512R1
- PSA_ECC_CURVE_CURVE25519
- PSA_ECC_CURVE_CURVE448
- PSA_ECC_CURVE_SECP160K1
- PSA_ECC_CURVE_SECP160R1
- PSA_ECC_CURVE_SECP160R2
- PSA_ECC_CURVE_SECP192K1
- PSA_ECC_CURVE_SECP192R1
- PSA_ECC_CURVE_SECP224K1
- PSA_ECC_CURVE_SECP224R1
- PSA_ECC_CURVE_SECP256K1
- PSA_ECC_CURVE_SECP256R1
- PSA_ECC_CURVE_SECP384R1
- PSA_ECC_CURVE_SECP521R1
- PSA_ECC_CURVE_SECT163K1
- PSA_ECC_CURVE_SECT163R1
- PSA_ECC_CURVE_SECT163R2
- PSA_ECC_CURVE_SECT193R1
- PSA_ECC_CURVE_SECT193R2
- PSA_ECC_CURVE_SECT233K1
- PSA_ECC_CURVE_SECT233R1
- PSA_ECC_CURVE_SECT239K1
- PSA_ECC_CURVE_SECT283K1
- PSA_ECC_CURVE_SECT283R1
- PSA_ECC_CURVE_SECT409K1
- PSA_ECC_CURVE_SECT409R1
- PSA_ECC_CURVE_SECT571K1
- PSA_ECC_CURVE_SECT571R1
- PSA_KEY_TYPE_GET_CURVE
- PSA_KEY_TYPE_GET_GROUP

— The following macros have been added:

- PSA_DH_FAMILY_RFC7919
- PSA_ECC_FAMILY_BRAINPOOL_P_R1
- PSA_ECC_FAMILY_SECP_K1

        - PSA_ECC_FAMILY_SECP_R1
        - PSA_ECC_FAMILY_SECP_R2
        - PSA_ECC_FAMILY_SECT_K1
        - PSA_ECC_FAMILY_SECT_R1
        - PSA_ECC_FAMILY_SECT_R2
        - PSA_ECC_FAMILY_MONTGOMERY
        - PSA_KEY_TYPE_DH_GET_FAMILY
        - PSA_KEY_TYPE_ECC_GET_FAMILY
    - The following macros have new values:
        - PSA_KEY_TYPE_AES
        - PSA_KEY_TYPE_ARC4
        - PSA_KEY_TYPE_CAMELLIA
        - PSA_KEY_TYPE_CHACHA20
        - PSA_KEY_TYPE_DERIVE
        - PSA_KEY_TYPE_DES
        - PSA_KEY_TYPE_HMAC
        - PSA_KEY_TYPE_NONE
        - PSA_KEY_TYPE_RAW_DATA
        - PSA_KEY_TYPE_RSA_KEY_PAIR
        - PSA_KEY_TYPE_RSA_PUBLIC_KEY
    - The following macros with specification-defined values have new example implementations:
        - PSA_BLOCK_CIPHER_BLOCK_LENGTH
        - PSA_KEY_TYPE_DH_KEY_PAIR
        - PSA_KEY_TYPE_DH_PUBLIC_KEY
        - PSA_KEY_TYPE_ECC_KEY_PAIR
        - PSA_KEY_TYPE_ECC_PUBLIC_KEY
        - PSA_KEY_TYPE_IS_ASYMMETRIC
        - PSA_KEY_TYPE_IS_DH
        - PSA_KEY_TYPE_IS_DH_KEY_PAIR
        - PSA_KEY_TYPE_IS_DH_PUBLIC_KEY
        - PSA_KEY_TYPE_IS_ECC
        - PSA_KEY_TYPE_IS_ECC_KEY_PAIR
        - PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY
        - PSA_KEY_TYPE_IS_KEY_PAIR
        - PSA_KEY_TYPE_IS_PUBLIC_KEY
        - PSA_KEY_TYPE_IS_RSA
        - PSA_KEY_TYPE_IS_UNSTRUCTURED
        - PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY
        - PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR
- Add ECC family PSA_ECC_FAMILY_FRP for the FRP256v1 curve.
- Restructure psa_algorithm_t encoding, to increase consistency across algorithm categories.

- Algorithms that include a hash operation all use the same structure to encode the hash algorithm. The following PSA_ALG_XXXX_GET_HASH() macros have all been replaced by a single macro PSA_ALG_GET_HASH():
    - PSA_ALG_HKDF_GET_HASH()
    - PSA_ALG_HMAC_GET_HASH()
    - PSA_ALG_RSA_OAEP_GET_HASH()
    - PSA_ALG_SIGN_GET_HASH()
    - PSA_ALG_TLS12_PRF_GET_HASH()
    - PSA_ALG_TLS12_PSK_TO_MS_GET_HASH()
- Stream cipher algorithm macros have been removed; the key type indicates which cipher to use. Instead of PSA_ALG_ARC4 and PSA_ALG_CHACHA20, use PSA_ALG_STREAM_CIPHER.

All of the other PSA_ALG_XXX macros have updated values or updated example implementations.

- The following macros have new values:
    - PSA_ALG_ANY_HASH
    - PSA_ALG_CBC_MAC
    - PSA_ALG_CBC_NO_PADDING
    - PSA_ALG_CBC_PKCS7
    - PSA_ALG_CCM
    - PSA_ALG_CFB
    - PSA_ALG_CHACHA20_POLY1305
    - PSA_ALG_CMAC
    - PSA_ALG_CTR
    - PSA_ALG_ECDH
    - PSA_ALG_ECDSA_ANY
    - PSA_ALG_FFDH
    - PSA_ALG_GCM
    - PSA_ALG_MD2
    - PSA_ALG_MD4
    - PSA_ALG_MD5
    - PSA_ALG_OFB
    - PSA_ALG_RIPEMD160
    - PSA_ALG_RSA_PKCS1V15_CRYPT
    - PSA_ALG_RSA_PKCS1V15_SIGN_RAW
    - PSA_ALG_SHA_1
    - PSA_ALG_SHA_224
    - PSA_ALG_SHA_256
    - PSA_ALG_SHA_384
    - PSA_ALG_SHA_512
    - PSA_ALG_SHA_512_224
    - PSA_ALG_SHA_512_256
    - PSA_ALG_SHA3_224
    - PSA_ALG_SHA3_256

    - PSA_ALG_SHA3_384
    - PSA_ALG_SHA3_512
    - PSA_ALG_XTS
  - The following macros with specification-defined values have new example implementations:
    - PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG()
    - PSA_ALG_AEAD_WITH_SHORTENED_TAG()
    - PSA_ALG_DETERMINISTIC_ECDSA()
    - PSA_ALG_ECDSA()
    - PSA_ALG_FULL_LENGTH_MAC()
    - PSA_ALG_HKDF()
    - PSA_ALG_HMAC()
    - PSA_ALG_IS_AEAD()
    - PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER()
    - PSA_ALG_IS_ASYMMETRIC_ENCRYPTION()
    - PSA_ALG_IS_BLOCK_CIPHER_MAC()
    - PSA_ALG_IS_CIPHER()
    - PSA_ALG_IS_DETERMINISTIC_ECDSA()
    - PSA_ALG_IS_ECDH()
    - PSA_ALG_IS_ECDSA()
    - PSA_ALG_IS_FFDH()
    - PSA_ALG_IS_HASH()
    - PSA_ALG_IS_HASH_AND_SIGN()
    - PSA_ALG_IS_HKDF()
    - PSA_ALG_IS_HMAC()
    - PSA_ALG_IS_KEY_AGREEMENT()
    - PSA_ALG_IS_KEY_DERIVATION()
    - PSA_ALG_IS_MAC()
    - PSA_ALG_IS_RANDOMIZED_ECDSA()
    - PSA_ALG_IS_RAW_KEY_AGREEMENT()
    - PSA_ALG_IS_RSA_OAEP()
    - PSA_ALG_IS_RSA_PKCS1V15_SIGN()
    - PSA_ALG_IS_RSA_PSS()
    - PSA_ALG_IS_SIGN()
    - PSA_ALG_IS_SIGN_MESSAGE()
    - PSA_ALG_IS_STREAM_CIPHER()
    - PSA_ALG_IS_TLS12_PRF()
    - PSA_ALG_IS_TLS12_PSK_TO_MS()
    - PSA_ALG_IS_WILDCARD()
    - PSA_ALG_KEY_AGREEMENT()
    - PSA_ALG_KEY_AGREEMENT_GET_BASE()
    - PSA_ALG_KEY_AGREEMENT_GET_KDF()
    - PSA_ALG_RSA_OAEP()

    - PSA_ALG_RSA_PKCS1V15_SIGN()
    - PSA_ALG_RSA_PSS()
    - PSA_ALG_TLS12_PRF()
    - PSA_ALG_TLS12_PSK_TO_MS()
    - PSA_ALG_TRUNCATED_MAC()
- Added ECB block cipher mode, with no padding, as PSA_ALG_ECB_NO_PADDING.
- Add functions to suspend and resume hash operations:
    - psa_hash_suspend() halts the current operation and outputs a hash suspend state.
    - psa_hash_resume() continues a previously suspended hash operation.

  The format of the hash suspend state is documented in *Hash suspend state* on page 120, and supporting macros are provided for using this API:
    - PSA_HASH_SUSPEND_OUTPUT_SIZE()
    - PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE
    - PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH
    - PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH()
    - PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH()
    - PSA_HASH_BLOCK_LENGTH()
- Complement PSA_ERROR_STORAGE_FAILURE with new error codes PSA_ERROR_DATA_CORRUPT and PSA_ERROR_DATA_INVALID. These permit an implementation to distinguish different causes of failure when reading from key storage.
- Added input step PSA_KEY_DERIVATION_INPUT_CONTEXT for key derivation, supporting obvious mapping from the step identifiers to common KDF constructions.

Clarifications

- Clarified rules regarding modification of parameters in concurrent environments.
- Guarantee that psa_destroy_key(PSA_KEY_ID_NULL) always returns PSA_SUCCESS.
- Clarified the TLS PSK to MS key agreement algorithm.
- Document the key policy requirements for all APIs that accept a key parameter.
- Document more of the error codes for each function.

Other changes

- Require C99 for this specification instead of C89.
- Removed references to non-standard mbed-crypto header files. The only header file that applications need to include is psa/crypto.h.
- Reorganized the API reference, grouping the elements in a more natural way.
- Improved the cross referencing between all of the document sections, and from code snippets to API element descriptions.

C.1.3 Changes between *1.0 beta 2* and *1.0 beta 3*

Changes to the API

- Change the value of error codes, and some names, to align with other PSA specifications. The name changes are:
  - PSA_ERROR_UNKNOWN_ERROR → PSA_ERROR_GENERIC_ERROR
  - PSA_ERROR_OCCUPIED_SLOT → PSA_ERROR_ALREADY_EXISTS
  - PSA_ERROR_EMPTY_SLOT → PSA_ERROR_DOES_NOT_EXIST
  - PSA_ERROR_INSUFFICIENT_CAPACITY → PSA_ERROR_INSUFFICIENT_DATA
  - PSA_ERROR_TAMPERING_DETECTED → PSA_ERROR_CORRUPTION_DETECTED
- Change the way keys are created to avoid "half-filled" handles that contained key metadata, but no key material. Now, to create a key, first fill in a data structure containing its attributes, then pass this structure to a function that both allocates resources for the key and fills in the key material. This affects the following functions:
  - psa_import_key(), psa_generate_key(), psa_generator_import_key() and psa_copy_key() now take an attribute structure, as a pointer to psa_key_attributes_t, to specify key metadata. This replaces the previous method of passing arguments to psa_create_key() or to the key material creation function or calling psa_set_key_policy().
  - psa_key_policy_t and functions operating on that type no longer exist. A key's policy is now accessible as part of its attributes.
  - psa_get_key_information() is also replaced by accessing the key's attributes, retrieved with psa_get_key_attributes().
  - psa_create_key() no longer exists. Instead, set the key id attribute and the lifetime attribute before creating the key material.
- Allow psa_aead_update() to buffer data.
- New buffer size calculation macros.
- Key identifiers are no longer specific to a given lifetime value. psa_open_key() no longer takes a lifetime parameter.
- Define a range of key identifiers for use by applications and a separate range for use by implementations.
- Avoid the unusual terminology "generator": call them "key derivation operations" instead. Rename a number of functions and other identifiers related to for clarity and consistency:
  - psa_crypto_generator_t → psa_key_derivation_operation_t
  - PSA_CRYPTO_GENERATOR_INIT → PSA_KEY_DERIVATION_OPERATION_INIT
  - psa_crypto_generator_init() → psa_key_derivation_operation_init()
  - PSA_GENERATOR_UNBRIDLED_CAPACITY → PSA_KEY_DERIVATION_UNLIMITED_CAPACITY
  - psa_set_generator_capacity() → psa_key_derivation_set_capacity()
  - psa_get_generator_capacity() → psa_key_derivation_get_capacity()
  - psa_key_agreement() → psa_key_derivation_key_agreement()
  - psa_generator_read() → psa_key_derivation_output_bytes()
  - psa_generate_derived_key() → psa_key_derivation_output_key()
  - psa_generator_abort() → psa_key_derivation_abort()

    - psa_key_agreement_raw_shared_secret() → psa_raw_key_agreement()
    - PSA_KDF_STEP_xxx → PSA_KEY_DERIVATION_INPUT_xxx
    - PSA_xxx_KEYPAIR → PSA_xxx_KEY_PAIR
- Convert TLS1.2 KDF descriptions to multi-part key derivation.

Clarifications

- Specify psa_generator_import_key() for most key types.
- Clarify the behavior in various corner cases.
- Document more error conditions.

C.1.4 Changes between *1.0 beta 1* and *1.0 beta 2*

Changes to the API

- Remove obsolete definition PSA_ALG_IS_KEY_SELECTION.
- PSA_AEAD_FINISH_OUTPUT_SIZE: remove spurious parameter plaintext_length.

Clarifications

- psa_key_agreement(): document alg parameter.

Other changes

- Document formatting improvements.

C.2 Planned changes for version 1.0.x

Future versions of this specification that use a 1.0.x version will describe the same API as this specification. Any changes will not affect application compatibility and will not introduce major features. These updates are intended to add minor requirements on implementations, introduce optional definitions, make corrections, clarify potential or actual ambiguities, or improve the documentation.

These are the changes that we are currently planning to make for version 1.0.x:

- Declare identifiers for additional cryptographic algorithms.
- Mandate certain checks when importing some types of asymmetric keys.
- Specify the computation of algorithm and key type values.
- Further clarifications on API usage and implementation.

C.3 Future additions

Major additions to the API will be defined in future drafts and editions of a 1.x or 2.x version of this specification. Features that are being considered include:

- Multi-part operations for hybrid cryptography. For example, this includes hash-and-sign for EdDSA, and hybrid encryption for ECIES.
- A more general interface to key derivation and key exchange. This would enable an application to derive a non-extractable session key from non-extractable secrets, without leaking the intermediate material.
- Key wrapping mechanisms to extract and import keys in an encrypted and authenticated form.
- Key discovery mechanisms. This would enable an application to locate a key by its name or attributes.
- Implementation capability description. This would enable an application to determine the algorithms, key types and storage lifetimes that the implementation provides.
- An ownership and access control mechanism allowing a multi-client implementation to have privileged clients that are able to manage keys of other clients.

Index of API elements

PSA_B

PSA_C

PSA_D

PSA_E

PSA_G

PSA_H

PSA_I

PSA_K

PSA_M

PSA_P

PSA_R

PSA_S

PSA_T

PSA_V