PSA Certified Crypto API 1.4

Document number:	IHI 0086
Release Quality:	Final
Issue Number:	0
Confidentiality:	Non-confidential
Date of Issue:	17/11/2025

Abstract

This document is part of the PSA Certified API specifications. It defines interfaces to provide cryptographic operations and key storage services.

Contents

About this document

Release information

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

Table 1 Document revision history

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 Certified API specifications.
May 2019	1.0 Beta 3	Non-confidential	Update for release with other PSA Certified 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.
February 2022	1.1.0 Final	Non-confidential	New API for EdDSA, password hashing and key stretching. Many significant clarifications and improvements across the documentation.
October 2022	1.1.1 Final	Non-confidential	Relicensed as open source under CC BY-SA 4.0. Improve support for TLS.
March 2023	1.1.2 Final	Non-confidential	Clarifications and fixes
February 2024	1.2.0 Final	Non-confidential	Better support for key agreement. New algorithms for Zigbee, XChaCha, TLS 1.2, and key derivation.
March 2024	1.2.1 Final	Non-confidential	Clarifications and fixes
March 2025	1.3.0 Final	Non-confidential	Integrated the PAKE extension. New API for key encapsulation. Support for additional key generation parameters.
June 2025	1.3.1 Final	Non-confidential	Clarifications and fixes
September 2025	1.3.2 Final	Non-confidential	GlobalPlatform governance of PSA Certified evaluation scheme.
November 2025	1.4.0 Final	Non-confidential	New algorithms for signatures with context, eXtended Output Functions, key wrapping, WPA3-SAE, and Ascon. Added key query and key registration functions.

The detailed changes in each release are described in *Document change history* on page 449.

PSA Certified Crypto API

License

Text and illustrations

About the license

The language in the additional patent license is largely identical to that in section 3 of the Apache License, Version 2.0 (Apache 2.0), with two exceptions:

1. Changes are made related to the defined terms, to align those defined terms with the terminology in CC BY-SA 4.0 rather than Apache 2.0 (for example, changing "Work" to "Licensed Material").
2. The scope of the defensive termination clause is changed from "any patent licenses granted to You" to "any licenses granted to You". This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

To view the full text of the Apache 2.0 license, visit apache.org/licenses/LICENSE-2.0.

Source code

References

This document refers to the following documents.

Table 2 Arm documents referenced by this document

Ref	Document Number	Title
[PSM]	ARM DEN 0128	*Platform Security Model*. developer.arm.com/documentation/den0128
[PSA-FFM]	ARM DEN 0063	*Arm® Platform Security Architecture Firmware Framework*. developer.arm.com/documentation/den0063
[PSA-STAT]	ARM IHI 0097	*PSA Certified Status code API*. arm-software.github.io/psa-api/status-code
[PSA-PQC]	ARM AES 0119	*PSA Certified Crypto API 1.4 PQC Extension*. arm-software.github.io/psa-api/crypto

Table 3 Other documents referenced by this document

Ref	Title
[C99]	ISO/IEC, *ISO/IEC 9899:1999 — Programming Languages — C*, December 1999. www.iso.org/standard/29237.html
[CHACHA20]	Bernstein, D., *ChaCha, a variant of Salsa20*, January 2008. http://cr.yp.to/chacha/chacha-20080128.pdf
[CLULOW]	Clulow, Jolyon, *On the Security of PKCS #11*, 2003. link.springer.com/chapter/10.1007/978-3-540-45238-6_32
[CSTC0002]	Cryptography Standardization Technical Committee, *GM/T 0002-2012: SM4 block cipher algorithm*, March 2012.
[CSTC0004]	Cryptography Standardization Technical Committee, *GM/T 0004-2012: SM3 cryptographic hash algorithm*, March 2012.
[Curve25519]	Bernstein et al., *Curve25519: new Diffie-Hellman speed records*, LNCS 3958, 2006. www.iacr.org/archive/pkc2006/39580209/39580209.pdf
[Curve448]	Hamburg, *Ed448-Goldilocks, a new elliptic curve*, NIST ECC Workshop, 2015. eprint.iacr.org/2015/625.pdf
[Ed25519]	Bernstein et al., *Twisted Edwards curves*, Africacrypt, 2008. eprint.iacr.org/2008/013.pdf
[Ed448]	Hamburg, *Ed448-Goldilocks, a new elliptic curve*, NIST ECC Workshop, 2015. eprint.iacr.org/2015/625.pdf
[FIPS180-4]	NIST, *FIPS Publication 180-4: Secure Hash Standard (SHS)*, August 2015. doi.org/10.6028/NIST.FIPS.180-4

continues on next page

Table 3 – continued from previous page

Ref	Title
[FIPS186-4]	NIST, *FIPS Publication 186-4: Digital Signature Standard (DSS)*, July 2013. doi.org/10.6028/NIST.FIPS.186-4
[FIPS197]	NIST, *FIPS Publication 197: Advanced Encryption Standard (AES)*, November 2001. doi.org/10.6028/NIST.FIPS.197
[FIPS202]	NIST, *FIPS Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*, August 2015. 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. www.ssi.gouv.fr/agence/rayonnement-scientifique/publications-scientifiques/articles-ouvrages-actes
[IEEE-802.11]	IEEE, *IEEE 802.11-2024: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, 2024. standards.ieee.org/ieee/802.11/10548/
[IEEE-CCM]	IEEE, *IEEE Standard for Low-Rate Wireless Networks*, 2020. standards.ieee.org/ieee/802.15.4/7029/
[IEEE-XTS]	IEEE, *1619-2018 — IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*, January 2019. ieeexplore.ieee.org/servlet/opac?punumber=8637986
[ISO10118]	ISO/IEC, *ISO/IEC 10118-3:2018 IT Security techniques — Hash-functions — Part 3: Dedicated hash-functions*, October 2018. www.iso.org/standard/67116.html
[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. www.iso.org/standard/50375.html
[MATTER]	CSA, *Matter Specification, Version 1.2*, October 2023. csa-iot.org/all-solutions/matter/
[NTT-CAM]	NTT Corporation and Mitsubishi Electric Corporation, *Specification of Camellia — a 128-bit Block Cipher*, September 2001. info.isl.ntt.co.jp/crypt/eng/camellia/specifications
[RFC1319]	IETF, *The MD2 Message-Digest Algorithm*, April 1992. tools.ietf.org/html/rfc1319.html
[RFC1320]	IETF, *The MD4 Message-Digest Algorithm*, April 1992. tools.ietf.org/html/rfc1320.html
[RFC1321]	IETF, *The MD5 Message-Digest Algorithm*, April 1992. tools.ietf.org/html/rfc1321.html
[RFC2104]	IETF, *HMAC: Keyed-Hashing for Message Authentication*, February 1997. tools.ietf.org/html/rfc2104.html
[RFC2315]	IETF, *PKCS #7: Cryptographic Message Syntax Version 1.5*, March 1998. 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. tools.ietf.org/html/rfc3279.html

continues on next page

Table 3 – continued from previous page

Ref	Title
[RFC3394]	IETF, *Advanced Encryption Standard (AES) Key Wrap Algorithm*, September 2002. tools.ietf.org/html/rfc3394.html
[RFC3526]	IETF, *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*, May 2003. tools.ietf.org/html/rfc3526.html
[RFC3610]	IETF, *Counter with CBC-MAC (CCM)*, September 2003. tools.ietf.org/html/rfc3610
[RFC3713]	IETF, *A Description of the Camellia Encryption Algorithm*, April 2004. tools.ietf.org/html/rfc3713
[RFC4279]	IETF, *Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*, December 2005. tools.ietf.org/html/rfc4279.html
[RFC4615]	IETF, *The Advanced Encryption Standard-Cipher-based Message Authentication Code-Pseudo-Random Function-128 (AES-CMAC-PRF-128) Algorithm for the Internet Key Exchange Protocol (IKE)*, August 2006. tools.ietf.org/html/rfc4615.html
[RFC5116]	IETF, *An Interface and Algorithms for Authenticated Encryption*, January 2008. tools.ietf.org/html/rfc5116.html
[RFC5246]	IETF, *The Transport Layer Security (TLS) Protocol Version 1.2*, August 2008. tools.ietf.org/html/rfc5246.html
[RFC5489]	IETF, *ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)*, March 2009. tools.ietf.org/html/rfc5489.html
[RFC5639]	IETF, *Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation*, March 2010. tools.ietf.org/html/rfc5639.html
[RFC5649]	IETF, *Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm*, August 2009. tools.ietf.org/html/rfc5649.html
[RFC5794]	IETF, *A Description of the ARIA Encryption Algorithm*, March 2010. datatracker.ietf.org/doc/html/rfc5794
[RFC5869]	IETF, *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)*, May 2010. tools.ietf.org/html/rfc5869.html
[RFC5915]	IETF, *Elliptic Curve Private Key Structure*, June 2010. tools.ietf.org/html/rfc5915.html
[RFC5958]	IETF, *Asymmetric Key Packages*, August 2010. tools.ietf.org/html/rfc5958.html
[RFC6979]	IETF, *Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)*, August 2013. tools.ietf.org/html/rfc6979.html
[RFC7748]	IETF, *Elliptic Curves for Security*, January 2016. tools.ietf.org/html/rfc7748.html
[RFC7919]	IETF, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, August 2016. tools.ietf.org/html/rfc7919.html
[RFC8017]	IETF, *PKCS #1: RSA Cryptography Specifications Version 2.2*, November 2016. tools.ietf.org/html/rfc8017.html
[RFC8018]	IETF, *PKCS #5: Password-Based Cryptography Specification Version 2.1*, January 2017. tools.ietf.org/html/rfc8018.html

continues on next page

Table 3 – continued from previous page

Ref	Title
[RFC8032]	IRTF, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, January 2017. tools.ietf.org/html/rfc8032.html
[RFC8235]	IETF, *Schnorr Non-interactive Zero-Knowledge Proof*, September 2017. tools.ietf.org/html/rfc8235.html
[RFC8236]	IETF, *J-PAKE: Password-Authenticated Key Exchange by Juggling*, September 2017. tools.ietf.org/html/rfc8236.html
[RFC8439]	IRTF, *ChaCha20 and Poly1305 for IETF Protocols*, June 2018. tools.ietf.org/html/rfc8439.html
[RFC9383]	IETF, *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol*, September 2023. tools.ietf.org/html/rfc9383.html
[RIPEMD]	Dobbertin, Bosselaers and Preneel, *RIPEMD-160: A Strengthened Version of RIPEMD*, April 1996. homes.esat.kuleuven.be/~bosselae/ripemd160.html
[SEC1]	Standards for Efficient Cryptography, *SEC 1: Elliptic Curve Cryptography*, May 2009. www.secg.org/sec1-v2.pdf
[SEC2]	Standards for Efficient Cryptography, *SEC 2: Recommended Elliptic Curve Domain Parameters*, January 2010. www.secg.org/sec2-v2.pdf
[SEC2v1]	Standards for Efficient Cryptography, *SEC 2: Recommended Elliptic Curve Domain Parameters, Version 1.0*, September 2000. www.secg.org/SEC2-Ver-1.0.pdf
[SP800-108]	NIST, *NIST Special Publication 800-108r1: Recommendation for Key Derivation Using Pseudorandom Functions*, August 2022. doi.org/10.6028/NIST.SP.800-108r1
[SP800-232]	NIST, *NIST Special Publication 800-232: Ascon-Based Lightweight Cryptography Standards for Constrained Devices*, August 2025. doi.org/10.6028/NIST.SP.800-232
[SP800-30]	NIST, *NIST Special Publication 800-30 Revision 1: Guide for Conducting Risk Assessments*, September 2012. doi.org/10.6028/NIST.SP.800-30r1
[SP800-38A]	NIST, *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation: Methods and Techniques*, December 2001. 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. 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. doi.org/10.6028/NIST.SP.800-38D
[SP800-38F]	NIST, *NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping*, December 2012. doi.org/10.6028/NIST.SP.800-38F

continues on next page

Table 3 – continued from previous page

Ref	Title
[SP800-56A]	NIST, *NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography*, April 2018. 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. doi.org/10.6028/NIST.SP.800-67r2
[SPAKE2P-2]	IETF, *SPAKE2+, an Augmented PAKE (Draft 02)*, December 2020. datatracker.ietf.org/doc/draft-bar-cfrg-spake2plus-02
[THREAD]	Thread Group, *Thread Specification 1.3.0*, July 2022. www.threadgroup.org/ThreadSpec
[TLS-ECJPAKE]	Cragie, Hao, *Elliptic Curve J-PAKE Cipher Suites for Transport Layer Security (TLS)*, June 2016. datatracker.ietf.org/doc/html/draft-cragie-tls-ecjpake-01
[X9-62]	ANSI, *Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)*. standards.globalspec.com/std/1955141/ANSI%20X9.62
[XCHACHA]	Arciszewski, *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305*, January 2020. datatracker.ietf.org/doc/html/draft-irtf-cfrg-xchacha-03
[ZIGBEE]	zigbee alliance, *zigbee Specification*, April 2017. csa-iot.org/wp-content/uploads/2022/01/docs-05-3474-22-0csg-zigbee-specification-1.pdf

Terms and abbreviations

This document uses the following terms and abbreviations.

Table 4 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.

continues on next page

Table 4 – continued from previous page

Term	Meaning
Caller isolation	Property of an implementation in which there are multiple application instances, with a security boundary between the application instances, as well as between the cryptoprocessor and the application instances. See *Optional isolation* on page 21.
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.
Cryptoprocessor isolation	Property of an implementation in which there is a security boundary between the application and the cryptoprocessor, but the cryptoprocessor does not communicate with other applications. See *Optional isolation* on page 21.
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.
Isolation	Property of an implementation in which there is a security boundary between the application and the cryptoprocessor. See *Optional isolation* on page 21.
IV	See *Initialization vector*.
KDF	See *Key Derivation Function*.
Key agreement	An algorithm for two or more parties to establish a common secret key.
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 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.

continues on next page

Table 4 – continued from previous page

Term	Meaning
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.
No isolation	Property of an implementation in which there is no security boundary between the application and the cryptoprocessor. See *Optional isolation* on page 21.
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. See *Key lifetimes* on page 90.
Post-Quantum Cryptography (PQC)	A cryptographic scheme that relies on mathematical problems that do not have efficient algorithms for either classical or quantum computing.
PQC	See *Post-Quantum Cryptography*.
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.
Volatile key	A key that has a short lifespan and is guaranteed not to exist after a restart of an application instance. See *Key lifetimes* on page 90.

Potential for change

The contents of this specification are stable for version 1.4.

The following may change in updates to the version 1.4 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 example.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.

Feedback

We welcome feedback on the PSA Certified API documentation.

If you have comments on the content of this book, visit github.com/arm-software/psa-api/issues to create a new issue at the PSA Certified API GitHub project. Give:

- The title (Crypto API).
- The number and issue (IHI 0086 1.4.0).
- The location in the document to which your comments apply.
- A concise explanation of your comments.

We also welcome general suggestions for additions and improvements.

1 Introduction

1.1 About Platform Security Architecture

This document is one of a set of resources provided by Arm that can help organizations develop products that meet the security requirements of GlobalPlatform's PSA Certified evaluation scheme on Arm-based platforms. The PSA Certified scheme provides a framework and methodology that helps silicon manufacturers, system software providers and OEMs to develop more secure products. Arm resources that support PSA Certified range from threat models, standard architectures that simplify development and increase portability, and open-source partnerships that provide ready-to-use software. You can read more about PSA Certified here at www.psacertified.org and find more Arm resources here at developer.arm.com/platform-security-resources and www.trustedfirmware.org.

1.2 About the Crypto API

The interface described in this document is a PSA Certified API, that provides a portable programming interface to cryptographic operations, and key storage functionality, 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.

You can find additional resources relating to the Crypto API here at arm-software.github.io/psa-api/crypto, and find other PSA Certified APIs here at arm-software.github.io/psa-api.

This document includes:

- A rationale for the design. See *Design goals* on page 21.
- A high-level overview of the functionality provided by the interface. See *Functionality overview* on page 24.
- A description of typical architectures of implementations for this specification. See *Sample architectures* on page 30.
- General considerations for implementers of this specification, and for applications that use the interface defined in this specification. See *Implementation considerations* on page 39 and *Usage considerations* on page 43.
- A detailed definition of the API. See *Library management reference* on page 45, *Key management reference* on page 49, and *Cryptographic operation reference* on page 130.

PSA Certified Crypto API 1.4 PQC Extension [PSA-PQC] is a companion document for version 1.4 of this specification. [PSA-PQC] defines an API for *Post-Quantum Cryptography* (PQC) algorithms. The PQC API is now at FINAL status, and will be included in a future version of the Crypto API specification.

In future, 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 caller isolation must document any implementation-specific extension of the API that enables frontend instances to share data in any form.

An isolated implementation that only has a single frontend provides *cryptoprocessor isolation*.

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 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 the Crypto 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, which begins with *Library management reference* on page 45.

Future additions on page 467 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. See *Key types* on page 25.
- The key identifier that the application uses to refer to the key. See *Key identifiers* on page 25.
- A lifetime that determines when the key material is destroyed, and where it is stored. See *Key lifetimes* on page 25.
- A policy that determines how the key can be used. See *Key policies* on page 26.

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

- `psa_import_key()`
- `psa_generate_key()`
- `psa_generate_key_custom()`
- `psa_key_derivation_output_key()`

- `psa_key_derivation_output_key_custom()`
- `psa_key_agreement()`
- `psa_encapsulate()`
- `psa_decapsulate()`
- `psa_pake_get_shared_key()`
- `psa_copy_key()`
- `psa_attach_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 types

Each cryptographic algorithm requires a key that has the right form, in terms of the size of the key material and its numerical properties. The key type and key size encode that information about a key, and determine whether the key is compatible with a cryptographic algorithm.

Additional non-cryptographic key types enable applications to store other secret values in the keystore.

See *Key types* on page 53.

3.2.2 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.

See *Key identifiers* on page 98.

3.2.3 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 90.

3.2.4 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 100.

3.2.5 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 Cryptographic operations

The API supports cryptographic operations through two kinds of interfaces:

- A *single-part* function performs a whole operation in a single function call. For example, compute, verify, encrypt or decrypt. See *Single-part Functions*.
- A *multi-part operation* is a set of functions that work with a stored operation state. This provides more control over operation configuration, piecewise processing of large input data, or handling for multi-step processes. See *Multi-part operations* on page 27.

Depending on the mechanism, one or both kind of interfaces may be provided.

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.

Single-part functions do not meet the needs of all use cases:

- 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.
- Password-authenticated key exchange (PAKE) is a multi-step process.

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, which is shown in Figure 1.

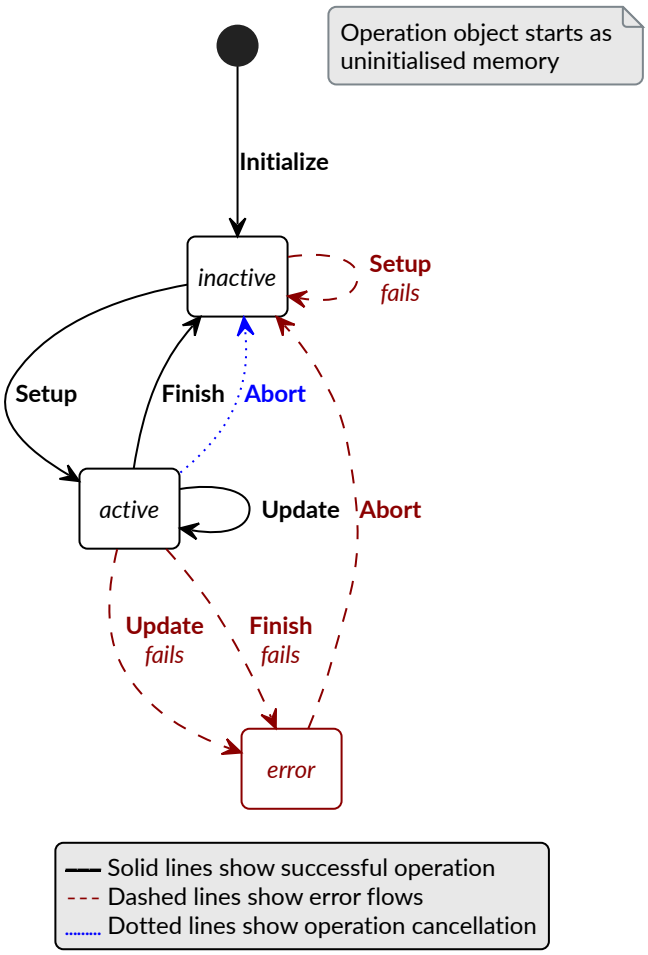

Figure 1 General state model for a multi-part operation

The typical sequence of actions with a multi-part operation is as follows:

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_OPERATION_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_OPERATION_INIT`.
   - Assign the result of calling the associated function `psa_xxx_operation_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. Each operation object defines one or more update functions, which are used to 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.
6. **Abort:** 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.

   `psa_xxx_abort()` can be called on an *inactive* operation, and this has no effect.

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* on page 38.
- 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 Symmetric cryptography

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

- Message digests, commonly known as hash functions. See *Message digests (Hashes)* on page 137.
- Message authentication codes (MAC). See *Message authentication codes (MAC)* on page 165.
- Symmetric ciphers. See *Unauthenticated ciphers* on page 181.
- Authenticated encryption with associated data (AEAD). See *Authenticated encryption with associated data (AEAD)* on page 207.
- Key derivation. See *Key derivation* on page 244.

Key derivation only provides multi-part operation, to support the flexibility required by these type of algorithms.

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.

3.3.4 Asymmetric cryptography

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

- Asymmetric encryption (also known as public-key encryption). See *Asymmetric encryption* on page 311.
- Asymmetric signature. See *Asymmetric signature* on page 278.
- Two-way key agreement (also known as key establishment). See *Key agreement* on page 317.
- Key encapsulation. See *Key encapsulation* on page 329.
- Password-authenticated key exchange (PAKE). See *Password-authenticated key exchange (PAKE)* on page 338.

For asymmetric encryption, the API provides *single-part* functions.

For asymmetric signature, the API provides single-part functions.

For key agreement, the API provides single-part functions and an additional input method for a key-derivation operation.

For key encapsulation, the API provides single-part functions.

For PAKE, the API provides a *multi-part* operation.

3.4 Randomness and key generation

We strongly recommended that implementations include a random generator, consisting of a cryptographically secure pseudorandom 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 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 *Cryptographic token and single-application processor* on page 31. 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 Header files

The header file for the Crypto API has the name `psa/crypto.h`. All of the API elements that are provided by an implementation must be visible to an application program that includes this header file.

```
#include "psa/crypto.h"
```

Implementations must provide their own version of the `psa/crypto.h` header file. Implementations can provide a subset of the API defined in this specification and a subset of the available algorithms. *Example header file* on page 392 provides an incomplete, example header file which includes all of the API elements. See also *Implementation considerations* on page 39.

The Crypto API uses the status code definitions that are shared with the other PSA Certified APIs. *PSA Certified Status code API* [PSA-STAT] defines these status codes in the `psa/error.h` header file. Applications

are not required to explicitly include the `psa/error.h` header file when using these status codes with the Crypto API. See *Status codes* on page 45.

5.2 API conventions

The interface in this specification is defined in terms of C macros, data types, and functions.

5.2.1 Identifier names

All of the identifiers defined in the Crypto API begin with the prefix `psa_`, for types and functions, or `PSA_` for macros.

Future versions of this specification will use the same prefix for additional API elements. It is recommended that applications and implementations do not use this prefix for their own identifiers, to avoid a potential conflict with a future version of the Crypto API.

5.2.2 Basic types

This specification makes use of standard C data types, including the fixed-width integer types from the ISO C99 specification update [C99]. The following standard C types are used:

`int32_t`	a 32-bit signed integer
`uint8_t`	an 8-bit unsigned integer
`uint16_t`	a 16-bit unsigned integer
`uint32_t`	a 32-bit unsigned integer
`uint64_t`	a 64-bit unsigned integer
`size_t`	an unsigned integer large enough to hold the size of an object in memory

5.2.3 Data types

Integral types are defined for specific API elements to provide clarity in the interface definition, and to improve code readability. For example, `psa_algorithm_t` and `psa_status_t`.

For enum-like integral types, the value `0` is usually reserved by the API to indicate an unspecified or invalid value.

Structure types are declared using `typedef` instead of a `struct` tag, also to improve code readability.

Fully-defined types must be declared exactly as defined in this specification. Types that are not fully defined in this specification must be defined by an implementation. See *Implementation-specific types* on page 39.

5.2.4 Constants

Constant values are defined using C macros. Constants defined in this specification have names that are all upper-case.

A constant macro evaluates to a compile-time constant expression.

5.2.5 Function-like macros

Function-like macros are C macros that take parameters, providing supporting functionality in the API. Function-like macros defined in this specification have names that are all upper-case.

Function-like macros are permitted to evaluate each argument multiple times or zero times. Providing arguments that have side effects will result in IMPLEMENTATION DEFINED behavior, and is non-portable.

If all of the arguments to a function-like macro are compile-time constant expressions, the then result evaluates to a compile-time constant expression.

If an argument to a function-like macro has an invalid value (for example, a value outside the domain of the function-like macro), then the result is IMPLEMENTATION DEFINED.

5.2.6 Functions

Functions defined in this specification have names that are all lower-case.

An implementation is permitted to declare any API function with `static inline` linkage, instead of the default `extern` linkage.

An implementation is permitted to also define a function-like macro with the same name as a function in this specification. If an implementation defines a function-like macro for a function from this specification, then:

- The implementation must also provide a definition of the function. This enables an application to take the address of a function defined in this specification.
- The function-like macro must expand to code that evaluates each of its arguments exactly once, as if the call was made to a C function. This enables an application to safely use arbitrary expressions as arguments to a function defined in this specification.

If a non-pointer argument to a function has an invalid value (for example, a value outside the domain of the function), then the function will normally return an error, as specified in the function definition. See also *Error handling*.

If a pointer argument to a function has an invalid value (for example, a pointer outside the address space of the program, or a null pointer), the result is IMPLEMENTATION DEFINED. See also *Pointer conventions* on page 36.

5.3 Error handling

5.3.1 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.

In particular, in the Crypto API, there are many conditions where the specification permits a function to return either `PSA_ERROR_INVALID_ARGUMENT` or `PSA_ERROR_NOT_SUPPORTED`. For example, `psa_hash_compute()` is passed a hash algorithm that the implementation does not support, it is IMPLEMENTATION DEFINED whether `PSA_ERROR_INVALID_ARGUMENT` or `PSA_ERROR_NOT_SUPPORTED` is returned.

Note:

This flexibility supports the scalability design goal. It permits implementations to not check whether unsupported algorithm identifier and key type values are valid or invalid.

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.

5.3.2 Behavior on error

In general, 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. Unless otherwise documented by the API or the implementation, the content of output parameters is not defined. 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 error state. The only valid action on an object in an error state is to abort it with the appropriate `psa_xxx_abort()` function. See *Multi-part operations* on page 27.
- 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.

Implementation note

When a function returns an error status, 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.

5.4 Parameter conventions

5.4.1 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 *Stability of parameters on page 37*.

5.4.2 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.

5.4.3 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.

5.4.4 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 *Stability of parameters*.

5.4.5 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.

5.5 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`.

5.5.1 Structure of key types and algorithms

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 algorithm or key category: hash, MAC, symmetric cipher, asymmetric encryption, and so on.
- The following bits identify a family of algorithms or keys 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.

The *Algorithm and key type encoding* on page 410 appendix provides a full definition of the encoding of key types and algorithm identifiers.

5.6 Concurrent calls

In some environments, an application can make calls to the 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.

Sequential consistency
: The result of two or more concurrent calls must be consistent with the same set of calls being executed sequentially in some order, provided that the calls 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 `psa_destroy_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.

The consistency requirement does not apply to errors that arise from resource failures or limitations. For example, errors resulting from resource exhaustion can arise in concurrent execution that do not arise in sequential execution.

As an example of this rule: suppose two calls are executed concurrently which both attempt to create a new key with the same key identifier that is not already in the key store. Then:

- If one call returns `PSA_ERROR_ALREADY_EXISTS`, then the other call must succeed.
- If one of the calls succeeds, then the other must fail: either with `PSA_ERROR_ALREADY_EXISTS` or some other error status.
- Both calls can fail with error codes that are not `PSA_ERROR_ALREADY_EXISTS`.

Parameter stability

If the application concurrently 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 427.
  - 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 permitted 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 non-volatile 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 (see *Memory cleanup*) for a *persistent key* 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 *Stability of parameters* on page 37.

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 with *no 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 the Crypto 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, the Crypto 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 *Concurrent calls* on page 38.

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 *Stability of parameters* on page 37.

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 Status codes

The Crypto API uses the status code definitions that are shared with the other PSA Certified APIs. The Crypto API also provides some Crypto API-specific status codes, see *Error codes specific to the Crypto API* on page 47.

The following elements are defined in `psa/error.h` from *PSA Certified Status code API* [PSA-STAT] (previously defined in [PSA-FFM]):

```
typedef int32_t psa_status_t;

#define PSA_SUCCESS ((psa_status_t)0)

#define PSA_ERROR_GENERIC_ERROR         ((psa_status_t)-132)
#define PSA_ERROR_NOT_PERMITTED         ((psa_status_t)-133)
#define PSA_ERROR_NOT_SUPPORTED         ((psa_status_t)-134)
#define PSA_ERROR_INVALID_ARGUMENT      ((psa_status_t)-135)
#define PSA_ERROR_INVALID_HANDLE        ((psa_status_t)-136)
#define PSA_ERROR_BAD_STATE             ((psa_status_t)-137)
#define PSA_ERROR_BUFFER_TOO_SMALL      ((psa_status_t)-138)
#define PSA_ERROR_ALREADY_EXISTS        ((psa_status_t)-139)
#define PSA_ERROR_DOES_NOT_EXIST        ((psa_status_t)-140)
#define PSA_ERROR_INSUFFICIENT_MEMORY   ((psa_status_t)-141)
#define PSA_ERROR_INSUFFICIENT_STORAGE  ((psa_status_t)-142)
#define PSA_ERROR_INSUFFICIENT_DATA     ((psa_status_t)-143)
#define PSA_ERROR_COMMUNICATION_FAILURE ((psa_status_t)-145)
#define PSA_ERROR_STORAGE_FAILURE       ((psa_status_t)-146)
#define PSA_ERROR_HARDWARE_FAILURE      ((psa_status_t)-147)
#define PSA_ERROR_INVALID_SIGNATURE     ((psa_status_t)-149)
#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)
```

These definitions must be available to an application that includes the `psa/crypto.h` header file.

Implementation note

An implementation is permitted to define the status code interface elements within the `psa/crypto.h` header file, or to define them via inclusion of a `psa/error.h` header file that is shared with the implementation of other PSA Certified APIs.

8.1.1 Common error codes

Some of the common status codes have a more precise meaning when returned by a function in the Crypto API, compared to the definitions in [PSA-STAT]. See also *Error handling* on page 34.

Error code	Meaning in the Crypto API
`PSA_ERROR_NOT_SUPPORTED`	[PSA-STAT] recommends the use of `PSA_ERROR_INVALID_ARGUMENT` for invalid parameter values. In the Crypto API, this is relaxed for algorithm identifier and key type parameters. It is recommended to return `PSA_ERROR_INVALID_ARGUMENT` for invalid values, but `PSA_ERROR_NOT_SUPPORTED` is also allowed, to permit implementations to avoid having to recognize all the cryptographic mechanisms that are defined in the PSA specification but not provided by that particular implementation.
`PSA_ERROR_INVALID_ARGUMENT`	[PSA-STAT] recommends the use of `PSA_ERROR_NOT_SUPPORTED` for unsupported parameter values. In the Crypto API, either `PSA_ERROR_INVALID_ARGUMENT` or `PSA_ERROR_NOT_SUPPORTED` can be returned when unsupported algorithm identifier or key type parameters are used. This allows implementations to avoid having to recognize all the cryptographic mechanisms that are defined in the PSA specification but not provided by that particular implementation.
`PSA_ERROR_INVALID_HANDLE`	A key identifier does not refer to an existing key. See also *Key identifiers* on page 25.
`PSA_ERROR_BAD_STATE`	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 can return this error if the caller has not initialized the library by a call to `psa_crypto_init()`.
`PSA_ERROR_BUFFER_TOO_SMALL`	Applications can call the `PSA_xxx_SIZE` macro listed in the function description to determine a sufficient buffer size.
`PSA_ERROR_STORAGE_FAILURE`	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.
`PSA_ERROR_CORRUPTION_DETECTED`	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.
`PSA_ERROR_DATA_CORRUPT`	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.

8.1.2 Error codes specific to the Crypto API

The following elements are defined in the `psa/crypto.h` header file.

`PSA_ERROR_INSUFFICIENT_ENTROPY` (macro)

A status code that indicates that 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_PADDING` (macro)

A status code that indicates that 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.

8.2 Crypto API library

8.2.1 API version

`PSA_CRYPTO_API_VERSION_MAJOR` (macro)

The major version of this implementation of the Crypto API.

```
#define PSA_CRYPTO_API_VERSION_MAJOR 1
```

`PSA_CRYPTO_API_VERSION_MINOR` (macro)

The minor version of this implementation of the Crypto API.

```
#define PSA_CRYPTO_API_VERSION_MINOR 4
```

8.2.2 Library initialization

`psa_crypto_init` (function)

Library initialization.

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

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

It is recommended that applications 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 any function that returns a `psa_status_t` result code before calling `psa_crypto_init()`, the following will occur:

- If initialization of the library is essential for secure operation of the function, the implementation must return `PSA_ERROR_BAD_STATE` or other appropriate error.
- If failure to initialize the library does not compromise the security of the function, the implementation must either provide the expected result for the function, or return `PSA_ERROR_BAD_STATE` or other appropriate error.

> **Note:**
>
> The following scenarios are examples where an implementation can require that the library has been initialized by calling `psa_crypto_init()`:
>
> - A client-server implementation, in which `psa_crypto_init()` establishes the communication with the server. No key management or cryptographic operation can be performed until this is done.
> - An implementation in which `psa_crypto_init()` initializes the random bit generator, and no operations that require the RNG can be performed until this is done. For example, random data, key, IV, or nonce generation; randomized signature or encryption; and algorithms that are implemented with blinding.

> **Warning**
>
> The set of functions that depend on successful initialization of the library is IMPLEMENTATION DEFINED. Applications that rely on calling functions before initializing the library might not be portable to other implementations.

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 98
- *Key lifetimes* on page 90
- *Key policies* on page 100

9.1.1 Managing key attributes

`psa_key_attributes_t` (typedef)

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 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 permits no usage except exporting a public key.
algorithm	PSA_ALG_NONE — which does not permit cryptographic usage, but permits 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. If the key is volatile in a non-default location, call psa_set_key_lifetime() to specify the location.
4. Set the key policy with psa_set_key_usage_flags() and psa_set_key_algorithm().
5. Set the key type with psa_set_key_type(). Skip this step if copying an existing key with psa_copy_key().
6. When generating a random key with psa_generate_key() or psa_generate_key_custom(), or deriving a key with psa_key_derivation_output_key() or psa_key_derivation_output_key_custom(), set the desired key size with psa_set_key_bits().
7. Call a key creation function: psa_import_key(), psa_generate_key(), psa_generate_key_custom(), psa_key_derivation_output_key(), psa_key_derivation_output_key_custom(), psa_key_agreement(), psa_encapsulate(), psa_decapsulate(), psa_pake_get_shared_key(), psa_copy_key(), or psa_attach_key(). This function reads the attribute object, creates a key with these attributes, and outputs an identifier for the newly created key.
8. 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`	Success. `attributes` contains the attributes of the key.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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` (typedef)

Encoding of a key type.

```
typedef uint16_t psa_key_type_t;
```

This is a structured bit field 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.

The *Algorithm and key type encoding* on page 410 appendix provides a full definition of the key type encoding.

`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

In the Crypto API, keys are typically used to store secrets that are specific to a set of related cryptographic algorithms. Keys can also be used to store non-cryptographic secrets or other data. The key type is used to identify what the key value is, and what can be used for.

- *Unstructured key types* on page 62 — defines types for non-key data and unstructured symmetric keys. For example, passwords, key-derivation secrets, or AES keys.
- *Structured key types* on page 72 — defines types for structured symmetric keys. For example, WPA3-SAE password tokens.
- *Asymmetric key types* on page 76 — defines types for asymmetric keys. For example, elliptic curve or SPAKE2+ keys.

`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: a value of type `psa_key_type_t`.

Description

This encompasses both symmetric keys and non-key data.

See *Unstructured key types* on page 62 for a list of unstructured 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: a value of type `psa_key_type_t`.

Description

See *Asymmetric key types* on page 76 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: a 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: a value of type `psa_key_type_t`.

9.2.3 Elliptic curve families

`psa_ecc_family_t` (typedef)

The type of identifiers of an elliptic curve family.

```
typedef uint8_t psa_ecc_family_t;
```

The curve family identifier is required to create a number of key types:

- ECC keys using `PSA_KEY_TYPE_ECC_KEY_PAIR()` or `PSA_KEY_TYPE_ECC_PUBLIC_KEY()`. These keys are used in various asymmetric signature, key-encapsulation, and key-agreement algorithms.
- SPAKE2+ keys using the `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()` or `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()`. These keys are used in the SPAKE2+ PAKE algorithms.
- WPA3-SAE password tokens using `PSA_KEY_TYPE_WPA3_SAE_ECC()`. These keys are used in the WPA3-SAE PAKE algorithms.

Elliptic curve family identifiers are also used to construct PAKE primitives for cipher suites based on elliptic curve groups. See *PAKE primitives* on page 338.

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` — Reserved. Not allocated to an elliptic curve family.

`0x01 - 0x7f` — Elliptic curve family identifiers defined by this standard. Unallocated values in this range are reserved for future use.

`0x80 - 0xff` — Invalid. Values in this range must not be used.

The least significant bit of a elliptic curve family identifier is a parity bit for the whole key type. See *Asymmetric key encoding* on page 424 for details of the encoding of asymmetric key types.

Implementation note

To provide other elliptic curve families, it is recommended that an implementation defines a key type with bit 15 set, which indicates an IMPLEMENTATION DEFINED key type.

`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 applications.

`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 applications.

`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 applications.

`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 applications.

`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`

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

`PSA_ECC_FAMILY_TWISTED_EDWARDS` (macro)

Twisted Edwards curves.

Added in version 1.1.

```
#define PSA_ECC_FAMILY_TWISTED_EDWARDS ((psa_ecc_family_t) 0x42)
```

This family comprises the following twisted Edwards curves:

- Edwards25519 : `key_bits = 255`. This curve is birationally equivalent to Curve25519.
- Edwards448 : `key_bits = 448`. This curve is birationally equivalent to Curve448.

Edwards25519 is defined in *Twisted Edwards curves* [Ed25519]. Edwards448 is defined in *Ed448-Goldilocks, a new elliptic curve* [Curve448].

9.2.4 Finite field Diffie-Hellman families

`psa_dh_family_t` (typedef)

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

```
typedef uint8_t psa_dh_family_t;
```

The group family identifier is required to create a number of key types:

- Diffie-Hellman keys using `PSA_KEY_TYPE_DH_KEY_PAIR()` or `PSA_KEY_TYPE_DH_PUBLIC_KEY()`. These keys are used in the FFDH key-agreement algorithm.
- WPA3-SAE password tokens using `PSA_KEY_TYPE_WPA3_SAE_DH()`. These keys are used in the WPA3-SAE PAKE algorithms.

Finite field Diffie-Hellman group identifiers are also used to construct PAKE primitives for cipher suites based on finite field groups. See *PAKE primitives* on page 338.

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

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

`0x00`
: Reserved. Not allocated to a Diffie-Hellman group family.

`0x01 - 0x7f`
: Diffie-Hellman group family identifiers defined by this standard. Unallocated values in this range are reserved for future use.

`0x80 - 0xff`
: Invalid. Values in this range must not be used.

The least significant bit of a finite field Diffie-Hellman group family identifier is a parity bit for the whole key type. See *Asymmetric key encoding* on page 424 for details of the encoding of asymmetric key types.

> **Implementation note**
>
> To provide other finite field Diffie-Hellman group families, it is recommended that an implementation defines a key type with bit 15 set, which indicates an IMPLEMENTATION DEFINED key type.

`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.

Groups in this family can 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_DH_FAMILY_RFC3526` (macro)

Finite field Diffie-Hellman groups defined for IKE2 in RFC 3526.

Added in version 1.4.

```
#define PSA_DH_FAMILY_RFC3526 ((psa_dh_family_t) 0x05)
```

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.

Groups in this family can be used with the `PSA_ALG_FFDH` key-agreement algorithm, or with the `PSA_ALG_WPA3_SAE_FIXED` and `PSA_ALG_WPA3_SAE_GDH` PAKE algorithms.

These groups are defined by *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)* [RFC3526].

9.2.5 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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

9.3 Unstructured key types

9.3.1 Non-key data

`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.

Compatible algorithms

A key of this type can also be used as a non-secret input to the following key-derivation algorithms:

- PSA_ALG_HKDF
- PSA_ALG_HKDF_EXPAND
- PSA_ALG_HKDF_EXTRACT
- PSA_ALG_SP800_108_COUNTER_HMAC
- PSA_ALG_SP800_108_COUNTER_CMAC
- PSA_ALG_TLS12_PRF
- PSA_ALG_TLS12_PSK_TO_MS

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

PSA_KEY_TYPE_DERIVE (macro)

A secret for key derivation.

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

This key type is for high-entropy secrets only. For low-entropy secrets, PSA_KEY_TYPE_PASSWORD should be used instead.

These keys can be used in the PSA_KEY_DERIVATION_INPUT_SECRET or PSA_KEY_DERIVATION_INPUT_PASSWORD input step of key-derivation algorithms.

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.

Compatible algorithms

A key of this type can be used as the secret input to the following key-derivation algorithms:

- PSA_ALG_HKDF
- PSA_ALG_HKDF_EXPAND
- PSA_ALG_HKDF_EXTRACT
- PSA_ALG_TLS12_PRF
- PSA_ALG_TLS12_PSK_TO_MS

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

PSA_KEY_TYPE_PASSWORD (macro)

A low-entropy secret for password hashing or key derivation.

Added in version 1.1.

```
#define PSA_KEY_TYPE_PASSWORD ((psa_key_type_t)0x1203)
```

This key type is suitable for passwords and passphrases which are typically intended to be memorizable by humans, and have a low entropy relative to their size. It can be used for randomly generated or derived keys with maximum or near-maximum entropy, but PSA_KEY_TYPE_DERIVE is more suitable for such keys. It is not suitable for passwords with extremely low entropy, such as numerical PINs.

These keys can be used in the PSA_KEY_DERIVATION_INPUT_PASSWORD input step of key-derivation algorithms. Algorithms that accept such an input were designed to accept low-entropy secret and are known as *password hashing* or *key stretching* algorithms.

These keys cannot be used in the PSA_KEY_DERIVATION_INPUT_SECRET input step of key-derivation algorithms, as the algorithms expect such an input to have high entropy.

The key policy determines which key-derivation algorithm the key can be used for, among the permissible subset defined above.

Compatible algorithms

A key of this type can be used as the password input to the following key-stretching algorithms:

- PSA_ALG_PBKDF2_HMAC
- PSA_ALG_PBKDF2_AES_CMAC_PRF_128

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

PSA_KEY_TYPE_PASSWORD_HASH (macro)

A secret value that can be used to verify a password hash.

Added in version 1.1.

```
#define PSA_KEY_TYPE_PASSWORD_HASH ((psa_key_type_t)0x1205)
```

The key policy determines which key-derivation algorithm the key can be used for, among the same permissible subset as for PSA_KEY_TYPE_PASSWORD.

Compatible algorithms

A key of this type can be used to output or verify the result of the following key-stretching algorithms:

- PSA_ALG_PBKDF2_HMAC
- PSA_ALG_PBKDF2_AES_CMAC_PRF_128

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

`PSA_KEY_TYPE_PEPPER` (macro)

A secret value that can be used when computing a password hash.

Added in version 1.1.

```
#define PSA_KEY_TYPE_PEPPER ((psa_key_type_t)0x1206)
```

The key policy determines which key-derivation algorithm the key can be used for, among the subset of algorithms that can use pepper.

Compatible algorithms

A key of this type can be used as the salt input to the following key-stretching algorithms:

- `PSA_ALG_PBKDF2_HMAC`
- `PSA_ALG_PBKDF2_AES_CMAC_PRF_128`

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

9.3.2 Symmetric cryptographic keys

`PSA_KEY_TYPE_HMAC` (macro)

HMAC key.

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

HMAC keys can be used in HMAC, or HMAC-based, algorithms. Although HMAC is parameterized by a specific hash algorithm, for example SHA-256, the hash algorithm is not specified in the key type. The permitted-algorithm policy for the key must specify a particular hash algorithm.

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, see *HMAC: Keyed-Hashing for Message Authentication* [RFC2104] §2.

It is recommended that an application does not construct HMAC keys that are longer than the block size of the hash algorithm that will be used. It is IMPLEMENTATION DEFINED whether an HMAC key that is longer than the hash block size is supported.

If the application does not control the length of the data used to construct the HMAC key, it is recommended that the application hashes the key data, when it exceeds the hash block length, before constructing the HMAC 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.

Compatible algorithms

- `PSA_ALG_HMAC`
- `PSA_ALG_SP800_108_COUNTER_HMAC` (secret input)

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

`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].

Compatible algorithms

- `PSA_ALG_CBC_MAC`
- `PSA_ALG_CMAC`
- `PSA_ALG_CTR`
- `PSA_ALG_CFB`
- `PSA_ALG_OFB`
- `PSA_ALG_XTS`

- PSA_ALG_CBC_NO_PADDING
- PSA_ALG_CBC_PKCS7
- PSA_ALG_ECB_NO_PADDING
- PSA_ALG_CCM
- PSA_ALG_GCM
- PSA_ALG_KW
- PSA_ALG_KWP
- PSA_ALG_SP800_108_COUNTER_CMAC (secret input)

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

PSA_KEY_TYPE_ARIA (macro)

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

Added in version 1.1.

```
#define PSA_KEY_TYPE_ARIA ((psa_key_type_t)0x2406)
```

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

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

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

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

The ARIA block cipher is defined in *A Description of the ARIA Encryption Algorithm* [RFC5794].

Compatible algorithms

- PSA_ALG_CBC_MAC
- PSA_ALG_CMAC
- PSA_ALG_CTR
- PSA_ALG_CFB
- PSA_ALG_OFB
- PSA_ALG_XTS
- PSA_ALG_CBC_NO_PADDING
- PSA_ALG_CBC_PKCS7
- PSA_ALG_ECB_NO_PADDING
- PSA_ALG_CCM

- PSA_ALG_GCM
- PSA_ALG_KW
- PSA_ALG_KWP
- PSA_ALG_SP800_108_COUNTER_CMAC (secret input)

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

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 applications.

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].

Compatible algorithms

- PSA_ALG_CBC_MAC
- PSA_ALG_CMAC
- PSA_ALG_CTR
- PSA_ALG_CFB
- PSA_ALG_OFB
- PSA_ALG_XTS
- PSA_ALG_CBC_NO_PADDING
- PSA_ALG_CBC_PKCS7
- PSA_ALG_ECB_NO_PADDING

Key format

The data format for import and export of the key is the raw bytes of the key. The parity bits in each 64-bit DES key element must be correct.

Key derivation

A call to `psa_key_derivation_output_key()` will construct a single 64-bit DES 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.

For 2-key 3DES and 3-key 3DES, this process is repeated to derive the 2nd and 3rd keys, as required.

`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].

Compatible algorithms

- `PSA_ALG_CBC_MAC`
- `PSA_ALG_CMAC`
- `PSA_ALG_CTR`
- `PSA_ALG_CFB`
- `PSA_ALG_OFB`
- `PSA_ALG_XTS`
- `PSA_ALG_CBC_NO_PADDING`
- `PSA_ALG_CBC_PKCS7`
- `PSA_ALG_ECB_NO_PADDING`
- `PSA_ALG_CCM`
- `PSA_ALG_GCM`

- PSA_ALG_KW
- PSA_ALG_KWP
- PSA_ALG_SP800_108_COUNTER_CMAC (secret input)

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

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 *GM/T 0002-2012: SM4 block cipher algorithm* [CSTC0002].

Compatible algorithms

- PSA_ALG_CBC_MAC
- PSA_ALG_CMAC
- PSA_ALG_CTR
- PSA_ALG_CFB
- PSA_ALG_OFB
- PSA_ALG_XTS
- PSA_ALG_CBC_NO_PADDING
- PSA_ALG_CBC_PKCS7
- PSA_ALG_ECB_NO_PADDING
- PSA_ALG_CCM
- PSA_ALG_GCM
- PSA_ALG_KW
- PSA_ALG_KWP
- PSA_ALG_SP800_108_COUNTER_CMAC (secret input)

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to psa_key_derivation_output_key() will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

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 applications.

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.

Compatible algorithms

- `PSA_ALG_STREAM_CIPHER`

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

`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.

Compatible algorithms

- `PSA_ALG_STREAM_CIPHER`
- `PSA_ALG_CHACHA20_POLY1305`

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw 32 bytes of output and use these as the key data.

`PSA_KEY_TYPE_XCHACHA20` (macro)

Key for the XChaCha20 stream cipher or the XChaCha20-Poly1305 AEAD algorithm.

Added in version 1.2.

```
#define PSA_KEY_TYPE_XCHACHA20 ((psa_key_type_t)0x2007)
```

The XChaCha20 key size is 256 bits (32 bytes).

- Use algorithm `PSA_ALG_STREAM_CIPHER` to use this key with the XChaCha20 cipher for unauthenticated encryption. See `PSA_ALG_STREAM_CIPHER` for details of this algorithm.
- Use algorithm `PSA_ALG_XCHACHA20_POLY1305` to use this key with the XChaCha20 cipher and Poly1305 authenticator for AEAD. See `PSA_ALG_XCHACHA20_POLY1305` for details of this algorithm.

Compatible algorithms

- `PSA_ALG_STREAM_CIPHER`
- `PSA_ALG_XCHACHA20_POLY1305`

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw 32 bytes of output and use these as the key data.

`PSA_KEY_TYPE_ASCON` (macro)

Key for the Ascon-AEAD128 AEAD algorithm.

Added in version 1.4.

```
#define PSA_KEY_TYPE_ASCON ((psa_key_type_t)0x2008)
```

The standard Ascon-AEAD128 key size is 128 bits (16 bytes).

For the nonce-masking variant of Ascon-AEAD128, use a key size of 256 bits (32-bytes).

See `PSA_ALG_ASCON_AEAD128` for details of this algorithm.

Compatible algorithms

- `PSA_ALG_ASCON_AEAD128`

Key format

The data format for import and export of the key is the raw bytes of the key.

Key derivation

A call to `psa_key_derivation_output_key()` will draw $m/8$ bytes of output and use these as the key data, where m is the bit-size of the key.

9.4 Structured key types

9.4.1 WPA3-SAE password tokens

The WPA3-SAE PAKE defines two techniques for generating the password element used during the PAKE protocol. The recommended hash-2-curve method is used to generate an intermediate password token, which is an element of the cyclic group used in the PAKE cipher suite. The password token can be stored as a key object, and later used in the PAKE operation when performing the WPA3-SAE protocol.

WPA3-SAE password tokens are defined for both elliptic curve and finite field groups.

See *WPA3-SAE password processing* on page 382.

`PSA_KEY_TYPE_WPA3_SAE_ECC` (macro)

WPA3-SAE password token using elliptic curves.

Added in version 1.4.

```
#define PSA_KEY_TYPE_WPA3_SAE_ECC(curve) /* specification-defined value */
```

Parameters

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

Description

The bit-size of a WPA3-SAE password token is the bit size associated with the specific curve within the elliptic curve family. See the documentation of the elliptic curve family for details.

To construct a WPA3-SAE password token, it must be output from key derivation operation using the `PSA_ALG_WPA3_SAE_H2E` algorithm.

> **Note:**
>
> To use a password token key with both `PSA_ALG_WPA3_SAE_FIXED` and `PSA_ALG_WPA3_SAE_GDH` algorithms, create the key with the wildcard `PSA_ALG_WPA3_SAE_ANY` permitted algorithm.

Compatible algorithms

- `PSA_ALG_WPA3_SAE_FIXED`
- `PSA_ALG_WPA3_SAE_GDH`

Key format

The password token is an element of the elliptic curve group, with value (x, y).

The data format for import and export of the password token is the concatenation of:

- x encoded as a big-endian m-byte string;
- y encoded as a big-endian m-byte string.

For an elliptic curve over $\mathbb{F}_p$, m is the integer for which $2^{8(m-1)} \leq p < 2^{8m}$.

> **Note:**
>
> This is the same format as the one used for group elements in the commit phase of the WPA3-SAE protocol, defined in [IEEE-802.11] §12.4.7.2.4.

Key derivation

A elliptic curve-based WPA3-SAE password token can only be derived using the PSA_ALG_WPA3_SAE_H2E algorithm. The call to psa_key_derivation_output_key() uses the method defined in [IEEE-802.11] §12.4.4.2.3 to generate the key value.

PSA_KEY_TYPE_WPA3_SAE_DH (macro)

WPA3-SAE password token using finite fields.

Added in version 1.4.

```
#define PSA_KEY_TYPE_WPA3_SAE_DH(group) /* specification-defined value */
```

Parameters

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

Description

The bit-size of the WPA3-SAE password token is the bit size associated with the specific group within the finite field Diffie-Hellman family. See the documentation of the selected Diffie-Hellman family for details.

To construct a WPA3-SAE password token, it must be output from key derivation operation using the PSA_ALG_WPA3_SAE_H2E algorithm.

> **Note:**
>
> To use a password token key with both PSA_ALG_WPA3_SAE_FIXED and PSA_ALG_WPA3_SAE_GDH algorithms, create the key with the wildcard PSA_ALG_WPA3_SAE_ANY permitted algorithm.

Compatible algorithms

- PSA_ALG_WPA3_SAE_FIXED
- PSA_ALG_WPA3_SAE_GDH

Key format

The password token is a finite-field group element $y \in [1, p-1]$, where p is the group's prime modulus.

The data format for import and export of the password token is y encoded as a big-endian m-byte string, where m is the integer for which $2^{8(m-1)} \leq p < 2^{8m}$.

> **Note:**
>
> This is the same format as the one used for group elements in the commit phase of the WPA3-SAE protocol, defined in [IEEE-802.11] §12.4.7.2.4.

Key derivation

A finite field-based WPA3-SAE password token can only be derived using the PSA_ALG_WPA3_SAE_H2E algorithm. The call to psa_key_derivation_output_key() uses the method defined in [IEEE-802.11] §12.4.4.3.3 to generate the key value.

`PSA_KEY_TYPE_IS_WPA3_SAE_ECC` (macro)

Whether a key type is a WPA3-SAE password token using elliptic curves.

Added in version 1.4.

```
#define PSA_KEY_TYPE_IS_WPA3_SAE_ECC(type) /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

`PSA_KEY_TYPE_WPA3_SAE_ECC_GET_FAMILY` (macro)

Extract the curve family from a WPA3-SAE password token using elliptic curves.

Added in version 1.4.

```
#define PSA_KEY_TYPE_WPA3_SAE_ECC_GET_FAMILY(type) \
    /* specification-defined value */
```

Parameters

`type`	A WPA3-SAE password token using elliptic curve key type: a value of type `psa_key_type_t` such that `PSA_KEY_TYPE_IS_WPA3_SAE_ECC(type)` is true.

Returns: `psa_ecc_family_t`

The elliptic curve family id, if `type` is a supported WPA3-SAE password token using elliptic curve key. Unspecified if `type` is not a supported WPA3-SAE password token using elliptic curve key.

`PSA_KEY_TYPE_IS_WPA3_SAE_DH` (macro)

Whether a key type is a WPA3-SAE password token using elliptic curves.

Added in version 1.4.

```
#define PSA_KEY_TYPE_IS_WPA3_SAE_DH(type) /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

`PSA_KEY_TYPE_WPA3_SAE_DH_GET_FAMILY` (macro)

Extract the finite field group family from a WPA3-SAE password token using finite fields.

Added in version 1.4.

```
#define PSA_KEY_TYPE_WPA3_SAE_DH_GET_FAMILY(type) \
    /* specification-defined value */
```

Parameters

`type`	A WPA3-SAE password token using finite fields key type: a value of type `psa_key_type_t` such that `PSA_KEY_TYPE_IS_WPA3_SAE_DH(type)` is true.

Returns: `psa_ecc_family_t`

The finite field group family id, if `type` is a supported WPA3-SAE password token using finite fields key. Unspecified if `type` is not a supported WPA3-SAE password token using finite fields key.

9.5 Asymmetric key types

Asymmetric keys come in pairs. One is a private or secret key, which must be kept confidential. The other is a public key, which is meant to be shared with other participants in the protocol.

> **Note:**
>
> Depending on the type of cryptographic scheme, the private key can be referred to as the *prover key* or the *decapsulation key*, and the public key can be referred to as the *verifier key* or the *encapsulation key*.

The Crypto API defines the following types of asymmetric key:

- *RSA keys*
- *Elliptic Curve keys* on page 78
- *Diffie Hellman keys* on page 84
- *SPAKE2+ keys* on page 86

In the Crypto API, key objects can either be a key pair, providing both the private and public key, or just a public key. The difference in the key type values for a key pair and a public key for the same scheme is common across all asymmetric keys.

The `PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY()` and `PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR()` macros convert from one type to the other.

9.5.1 RSA keys

`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)
```

The size of an RSA key is the bit size of the modulus.

Compatible algorithms

- `PSA_ALG_RSA_OAEP`
- `PSA_ALG_RSA_PKCS1V15_CRYPT`
- `PSA_ALG_RSA_PKCS1V15_SIGN`

- PSA_ALG_RSA_PKCS1V15_SIGN_RAW
- PSA_ALG_RSA_PSS
- PSA_ALG_RSA_PSS_ANY_SALT

Key format

The data format for import and export of a key-pair 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)
    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.

See PSA_KEY_TYPE_RSA_PUBLIC_KEY for the data format used when exporting the public key with `psa_export_public_key()`.

Key generation

A call to `psa_generate_key()` will generate an RSA key-pair with the default public exponent of `65537`. The modulus $n = pq$ is a product of two probabilistic primes p and q, where $2^{r-1} \leq n < 2^r$ and r is the bit size specified in the attributes.

The exponent can be explicitly specified in non-default production parameters in a call to `psa_generate_key_custom()`. Use the following custom production parameters:

- The production parameters structure, `custom`, must have `flags` set to zero.
- If `custom_data_length == 0`, the default exponent value `65537` is used.
- The additional production parameter buffer `custom_data` is the public exponent, in little-endian byte order.

  The exponent must be an odd integer greater than `1`. An implementation must support an exponent of `65537`, and is recommended to support an exponent of `3`, and can support other values.

  The maximum supported exponent value is IMPLEMENTATION DEFINED.

Key derivation

The method used by `psa_key_derivation_output_key()` to derive an RSA key-pair is *implementation defined*.

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (macro)

RSA public key.

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

The size of an RSA key is the bit size of the modulus.

Compatible algorithms

- `PSA_ALG_RSA_OAEP` (encryption only)
- `PSA_ALG_RSA_PKCS1V15_CRYPT` (encryption only)
- `PSA_ALG_RSA_PKCS1V15_SIGN` (signature verification only)
- `PSA_ALG_RSA_PKCS1V15_SIGN_RAW` (signature verification only)
- `PSA_ALG_RSA_PSS` (signature verification only)
- `PSA_ALG_RSA_PSS_ANY_SALT` (signature verification only)

Key format

The data format for import and export of a public key is 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
```

`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: a value of type `psa_key_type_t`.

9.5.2 Elliptic Curve keys

Elliptic curve keys are grouped into families of related curves. A keys for a specific curve is specified by a combination of the elliptic curve family and the bit-size of the key.

There are three categories of elliptic curve key, shown in Table 6 on page 79. The curve type affects the key format, the key-derivation procedure, and the algorithms which the key can be used with.

Table 6 Types of elliptic curve key

Curve type	Curve families
Weierstrass	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_BRAINPOOL_P_R1 PSA_ECC_FAMILY_FRP
Montgomery	PSA_ECC_FAMILY_MONTGOMERY
Twisted Edwards	PSA_ECC_FAMILY_TWISTED_EDWARDS

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.

Description

The size of an elliptic curve key is the bit size associated with the curve, that is, the bit size of q for a curve over a field $\mathbb{F}_q$. See the documentation of each elliptic curve family for details.

Compatible algorithms

Table 7 shows the compatible algorithms for each type of elliptic curve key-pair.

Table 7 Compatible algorithms for elliptic curve key-pairs

Curve type	Compatible algorithms
Weierstrass	Weierstrass curve key-pairs can be used in asymmetric signature, key-agreement, and key-encapsulation algorithms. PSA_ALG_DETERMINISTIC_ECDSA PSA_ALG_ECDSA PSA_ALG_ECDSA_ANY PSA_ALG_ECDH PSA_ALG_ECIES_SEC1

continues on next page

Table 7 – continued from previous page

Curve type	Compatible algorithms

Montgomery	Montgomery curve key-pairs can be used in key-agreement and key-encapsulation algorithms.
	`PSA_ALG_ECDH`
	`PSA_ALG_ECIES_SEC1`

Twisted Edwards	Twisted Edwards curve key-pairs can only be used in asymmetric signature algorithms. `PSA_ALG_PURE_EDDSA` `PSA_ALG_ED25519PH` (Edwards25519 only) `PSA_ALG_ED448PH` (Edwards448 only)

Key format

The data format for import and export of the key-pair depends on the type of elliptic curve. Table 8 shows the format for each type of elliptic curve key-pair.

See `PSA_KEY_TYPE_ECC_PUBLIC_KEY` for the data format used when exporting the public key with `psa_export_public_key()`.

Table 8 Key-pair formats for elliptic curve keys

Curve type	Key-pair format
Weierstrass	The key data is the content of the `privateKey` field of the `ECPrivateKey` format defined by *Elliptic Curve Private Key Structure* [RFC5915]. This is a $\lceil m/8 \rceil$-byte string in big-endian order, where m is the key size in bits.
Montgomery	The key data is 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 $\lceil m/8 \rceil$-byte string where m is the key size in bits. This is 32 bytes for Curve25519, and 56 bytes for Curve448.
Twisted Edwards	The key data is the private key, as defined by *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032]. This is a 32-byte string for Edwards25519, and a 57-byte string for Edwards448.

Key derivation

The key-derivation method used when calling `psa_key_derivation_output_key()` depends on the type of elliptic curve. Table 9 on page 81 shows the derivation method for each type of elliptic curve key.

Table 9 Key derivation for elliptic curve keys

Curve type	Key derivation
Weierstrass	A Weierstrass elliptic curve private key is $d \in [1, N-1]$, where N is the order of the curve's base point for ECC. Let m be the bit size of N, such that $2^{m-1} \leq N < 2^m$. This function generates the private key using the following process: 1. Draw a byte string of length $\lceil m/8 \rceil$ bytes. 2. If m is not a multiple of 8, set the most significant $8 * \lceil m/8 \rceil - m$ 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 $d = k + 1$ as the private key. This method allows compliance to NIST standards, specifically the methods titled *Key-Pair Generation by Testing Candidates* in [SP800-56A] §5.6.1.2.2 or *FIPS Publication 186-4: Digital Signature Standard (DSS)* [FIPS186-4] §B.4.2.
Montgomery	Draw a byte string whose length is determined by the curve, and set 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.
Twisted Edwards	Draw a byte string whose length is determined by the curve, and use this as the private key. That is: • Ed25519 (`PSA_ECC_FAMILY_MONTGOMERY`, 255 bits): draw a 32-byte string. • Ed448 (`PSA_ECC_FAMILY_MONTGOMERY`, 448 bits): draw a 57-byte string.

`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.

Description

The size of an elliptic curve public key is the same as the corresponding private key. See `PSA_KEY_TYPE_ECC_KEY_PAIR()` and the documentation of each elliptic curve family for details.

Compatible algorithms

Table 10 shows the compatible algorithms for each type of elliptic curve public key.

> **Note:**
>
> For key agreement, the public key of the peer is provided to the Crypto API as a buffer. This avoids the need to import the public-key data that is received from the peer, just to carry out the key-agreement algorithm.

Table 10 Compatible algorithms for elliptic curve public keys

Curve type	Compatible algorithms

Weierstrass	Weierstrass curve public keys can be used in asymmetric signature and key-encapsulation algorithms.
	`PSA_ALG_DETERMINISTIC_ECDSA`
	`PSA_ALG_ECDSA`
	`PSA_ALG_ECDSA_ANY`
	`PSA_ALG_ECIES_SEC1`

Montgomery	Montgomery curve public keys can only be used in key-encapsulation algorithms. `PSA_ALG_ECIES_SEC1`
Twisted Edwards	Twisted Edwards curve public keys can only be used in asymmetric signature algorithms. `PSA_ALG_PURE_EDDSA` `PSA_ALG_ED25519PH` (Edwards25519 only) `PSA_ALG_ED448PH` (Edwards448 only)

Key format

The data format for import and export of the public key depends on the type of elliptic curve. Table 11 on page 83 shows the format for each type of elliptic curve public key.

Table 11 Public-key formats for elliptic curve keys

Curve type	Public-key format
Weierstrass	The key data is 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 $\mathbb{F}_q$, then the representation of point P consists of: • The byte `0x04`; • x_P as a $\lceil m/8 \rceil$-byte string, big-endian; • y_P as a $\lceil m/8 \rceil$-byte string, big-endian.
Montgomery	The key data is the scalar value of the 'public key' in little-endian order as defined by *Elliptic Curves for Security* [RFC7748] §6. This is a $\lceil m/8 \rceil$-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)`.
Twisted Edwards	The key data is the public key, as defined by *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032]. This is a 32-byte string for Edwards25519, and a 57-byte string for Edwards448.

`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: a 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: a 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: a 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: a 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.

9.5.3 Diffie Hellman keys

`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 finite field Diffie-Hellman group family to be used.

Compatible algorithms

- `PSA_ALG_FFDH`

Key format

The data format for import and export of the key pair is the representation of the private key x as a big-endian byte string. The length of the byte string is the private key's size in bytes, and leading zeroes are not stripped.

See `PSA_KEY_TYPE_DH_PUBLIC_KEY` for the data format used when exporting the public key with `psa_export_public_key()`.

Key derivation

A call to `psa_key_derivation_output_key()` will use the following process, defined in *Key-Pair Generation by Testing Candidates* in *NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography* [SP800-56A] §5.6.1.1.4.

A finite field Diffie-Hellman private key is $x \in [1, p-1]$, where p is the group's prime modulus. Let m be the bit size of p, such that $2^{m-1} \leq p < 2^m$.

This function generates the private key using the following process:

1. Draw a byte string of length $\lceil m/8 \rceil$ bytes.

2. If m is not a multiple of 8, set the most significant $8 * \lceil m/8 \rceil - m$ 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 $k > p - 2$, discard the result and return to step 1.
5. Output $x = k + 1$ as the private key.

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 finite field Diffie-Hellman group family to be used.

Compatible algorithms

None: Finite field Diffie-Hellman public keys are exported to use in a key-agreement algorithm, and the peer key is provided to the PSA_ALG_FFDH key-agreement algorithm as a buffer of key data.

Key format

The data format for export of the public key 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.

PSA_KEY_TYPE_IS_DH (macro)

Whether a key type is a finite field 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: a value of type psa_key_type_t.

PSA_KEY_TYPE_IS_DH_KEY_PAIR (macro)

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

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

Parameters

type	A key type: a value of type psa_key_type_t.

PSA_KEY_TYPE_IS_DH_PUBLIC_KEY (macro)

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

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

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

`PSA_KEY_TYPE_DH_GET_FAMILY` (macro)

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

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

Parameters

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

Returns: `psa_dh_family_t`

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

9.5.4 SPAKE2+ keys

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR` (macro)

SPAKE2+ key pair: both the prover and verifier key.

Added in version 1.2.

```
#define PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(curve) /* specification-defined value */
```

Parameters

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

Description

The bit-size of a SPAKE2+ key is the size associated with the elliptic curve group, that is, $\lceil log_2(q) \rceil$ for a curve over a field $\mathbb{F}_q$. See *Elliptic Curve keys* on page 78 for details of each elliptic curve family.

To create a new SPAKE2+ key pair, use `psa_key_derivation_output_key()` as described in *SPAKE2+ registration* on page 372. The SPAKE2+ protocol recommends that a key-stretching key-derivation function, such as PBKDF2, is used to hash the SPAKE2+ password. This follows the recommended process described in [RFC9383].

A SPAKE2+ key pair can also be imported from a previously exported SPAKE2+ key pair.

The corresponding public key can be exported using `psa_export_public_key()`. See also `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()`.

Compatible algorithms

- `PSA_ALG_SPAKE2P_HMAC`
- `PSA_ALG_SPAKE2P_CMAC`
- `PSA_ALG_SPAKE2P_MATTER`

Key format

A SPAKE2+ key pair consists of the two values $w0$ and $w1$, which result from the SPAKE2+ registration phase, see *SPAKE2+ registration* on page 372. $w0$ and $w1$ are scalars in the same range as an elliptic curve private key from the group used as the SPAKE2+ primitive group.

The data format for import and export of the key pair is the concatenation of the formatted values for $w0$ and $w1$, using the standard formats for elliptic curve keys used by the Crypto API. For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_key()` would be the concatenation of:

- The P-256 private key $w0$. This is a 32-byte big-endian encoding of the integer $w0$.
- The P-256 private key $w1$. This is a 32-byte big-endian encoding of the integer $w1$.

See `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` for the data format used when exporting the public key with `psa_export_public_key()`.

Key derivation

A call to `psa_key_derivation_output_key()` will use the following process, which follows the recommendations for the registration process in *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383], and matches the specification of this process in *Matter Specification, Version 1.2* [MATTER].

The derivation of SPAKE2+ keys extracts $\lceil log_2(p)/8 \rceil + 8$ bytes from the PBKDF for each of $w0s$ and $w1s$, where p is the prime factor of the order of the elliptic curve group. The following sizes are used for extracting $w0s$ and $w1s$, depending on the elliptic curve:

- P-256: 40 bytes
- P-384: 56 bytes
- P-521: 74 bytes
- edwards25519: 40 bytes
- edwards448: 64 bytes

The calculation of $w0$, $w1$, and L then proceeds as described in [RFC9383].

> **Implementation note**
>
> The values of $w0$ and $w1$ are required as part of the SPAKE2+ key pair.
>
> It is IMPLEMENTATION DEFINED whether L is computed during key derivation, and stored as part of the key pair; or only computed when required from the key pair.

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (macro)

SPAKE2+ public key: the verifier key.

Added in version 1.2.

```
#define PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(curve) \
    /* specification-defined value */
```

Parameters

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

Description

The bit-size of an SPAKE2+ public key is the same as the corresponding private key. See `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()` and the documentation of each elliptic curve family for details.

To construct a SPAKE2+ public key, it must be imported.

Compatible algorithms

- `PSA_ALG_SPAKE2P_HMAC` (verification only)
- `PSA_ALG_SPAKE2P_CMAC` (verification only)
- `PSA_ALG_SPAKE2P_MATTER` (verification only)

Key format

A SPAKE2+ public key consists of the two values $w0$ and L, which result from the SPAKE2+ registration phase, see *SPAKE2+ registration* on page 372. $w0$ is a scalar in the same range as a elliptic curve private key from the group used as the SPAKE2+ primitive group. L is a point on the curve, similar to a public key from the same group, corresponding to the $w1$ value in the key pair.

The data format for import and export of the public key is the concatenation of the formatted values for $w0$ and L, using the standard formats for elliptic curve keys used by the Crypto API. For example, for SPAKE2+ over P-256 (secp256r1), the output from `psa_export_public_key()` would be the concatenation of:

- The P-256 private key $w0$. This is a 32-byte big-endian encoding of the integer $w0$.
- The P-256 public key L. This is a 65-byte concatenation of:
  - The byte `0x04`.
  - The 32-byte big-endian encoding of the x-coordinate of L.
  - The 32-byte big-endian encoding of the y-coordinate of L.

`PSA_KEY_TYPE_IS_SPAKE2P` (macro)

Whether a key type is a SPAKE2+ key, either a key pair or a public key.

Added in version 1.2.

```
#define PSA_KEY_TYPE_IS_SPAKE2P(type) /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

`PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR` (macro)

Whether a key type is a SPAKE2+ key pair.

Added in version 1.2.

```
#define PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR(type) \
    /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

`PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY` (macro)

Whether a key type is a SPAKE2+ public key.

Added in version 1.2.

```
#define PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY(type) \
    /* specification-defined value */
```

Parameters

`type`	A key type: a value of type `psa_key_type_t`.

`PSA_KEY_TYPE_SPAKE2P_GET_FAMILY` (macro)

Extract the curve family from a SPAKE2+ key type.

Added in version 1.2.

```
#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) /* specification-defined value */
```

Parameters

`type`	A SPAKE2+ key type: a value of type `psa_key_type_t` such that `PSA_KEY_TYPE_IS_SPAKE2P(type)` is true.

Returns: `psa_ecc_family_t`

The elliptic curve family id, if `type` is a supported SPAKE2+ key. Unspecified if `type` is not a supported SPAKE2+ key.

9.5.5 Support macros

`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.

9.6 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*.

9.6.1 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.

Volatile keys have the persistence level `PSA_KEY_PERSISTENCE_VOLATILE` in the key lifetime value, see *Key lifetime encoding* on page 91. Unless the key lifetime is explicitly set in the key attributes before creating a key, a volatile key will be created with the default `PSA_KEY_LIFETIME_VOLATILE` lifetime value.

To create a volatile key:

1. Populate a `psa_key_attributes_t` object with the required type, size, policy and other key attributes.
2. If a non-default storage location is being used, set the key lifetime in the attributes object.
3. 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.

9.6.2 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 42 and *Managing key material* on page 42.

9.6.3 Key lifetime encoding

`psa_key_lifetime_t` (typedef)

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 (typedef)

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 Crypto API.

Values for persistence levels defined by Crypto API are shown in Table 12.

Table 12 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.

continues on next page

Table 12 – continued from previous page

Persistence level	Definition
`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 Certified API-specified lifetime. The Crypto API does not define the meaning of these values, but another PSA Certified API may do so.
`128 - 254`	Persistent key with a vendor-specified lifetime. No PSA Certified API 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 Crypto 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` (typedef)

Encoding of key location indicators.

```
typedef uint32_t psa_key_location_t;
```

If an implementation of the Crypto 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 13 on page 94.

Table 13 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 Crypto API does not currently assign any meaning to these locations, but future versions of this specification or other PSA Certified APIs may do so.
`0x800000 - 0xffffff`	Vendor-defined locations. No PSA Certified API 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.

9.6.4 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 Crypto API implementation. 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.

9.6.5 Attribute accessors

`psa_set_key_lifetime` (function)

Set the lifetime of a key, for a persistent key or a non-default location.

```
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 a volatile lifetime (such that `PSA_KEY_LIFETIME_IS_VOLATILE(lifetime)` is true), 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 specific 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_generate_key_custom()`, `psa_key_derivation_output_key()`, `psa_key_derivation_output_key_custom()`, `psa_key_agreement()`, `psa_encapsulate()`, `psa_decapsulate()`, `psa_pake_get_shared_key()`, or `psa_copy_key()`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

9.6.6 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: a 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: a 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: a 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: a value of type `psa_key_persistence_t`.
`location`	The location indicator: a value of type `psa_key_location_t`.

Returns

The constructed lifetime value.

9.7 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.

9.7.1 Key identifier type

`psa_key_id_t` (typedef)

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 98.

`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)
```

9.7.2 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, this function sets the persistence level in the lifetime attribute to `PSA_KEY_PERSISTENCE_DEFAULT` without changing the key location. See *Key lifetimes* on page 90.

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_generate_key_custom()`, `psa_key_derivation_output_key()`, `psa_key_derivation_output_key_custom()`, `psa_key_agreement()`, `psa_encapsulate()`, `psa_decapsulate()`, `psa_pake_get_shared_key()`, or `psa_copy_key()`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

9.8 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 102.
- 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.

9.8.1 Permitted algorithms

The permitted algorithm is encoded using a algorithm identifier, as described in *Algorithms* on page 130.

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

If the permitted algorithm is `PSA_ALG_NONE`, no cryptographic operation is permitted with the key. The key can still be used for non-cryptographic actions such as exporting, if permitted by the usage flags.

For a cryptographic operation, the permitted algorithm value must exactly match the requested algorithm, except in the following cases:

- The following pairs of signature algorithms are considered equivalent for verification, but not for computing the signature:
  - `PSA_ALG_ECDSA` and `PSA_ALG_DETERMINISTIC_ECDSA`.
- A signature algorithm constructed with `PSA_ALG_ANY_HASH` permits the specified signature scheme with any hash algorithm. In addition, `PSA_ALG_RSA_PKCS1V15_SIGN(PSA_ALG_ANY_HASH)` also permits the `PSA_ALG_RSA_PKCS1V15_SIGN_RAW` signature algorithm.
- A standalone key-agreement algorithm also permits the specified key-agreement scheme to be combined with any key-derivation algorithm.
- An algorithm built from `PSA_ALG_AT_LEAST_THIS_LENGTH_MAC()` permits any MAC algorithm from the same base class (for example, CMAC) which computes or verifies a MAC length greater than or equal to the length encoded in the wildcard algorithm.
- An algorithm built from `PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG()` permits any AEAD algorithm from the same base class (for example, CCM) which computes or verifies a tag length greater than or equal to the length encoded in the wildcard algorithm.
- The `PSA_ALG_CCM_STAR_ANY_TAG` wildcard algorithm permits the `PSA_ALG_CCM_STAR_NO_TAG` cipher algorithm, the `PSA_ALG_CCM` AEAD algorithm, and the `PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CCM, tag_length)` truncated-tag AEAD algorithm for `tag_length` equal to 4, 8 or 16.
- The wildcard key policy `PSA_ALG_WPA3_SAE_ANY` permits a password key or WPA3-SAE password token key to be used with any WPA3-SAE cipher suite.

When a key is used in a cryptographic operation, the application supplies the algorithm to use for the operation. The algorithm and operation are 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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

9.8.2 Key usage flags

The usage flags are encoded in a bitmask, which has the type `psa_key_usage_t`. There are two kinds of usage flag:

1. Key-management usage flags.

- The extractable flag PSA_KEY_USAGE_EXPORT determines whether the key material can be extracted from the cryptoprocessor, or copied outside of its current security boundary.
- 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 42.

2. Cryptographic-operation usage flags.

   The following usage flags determine whether the corresponding cryptographic operations are permitted with the key:

   - PSA_KEY_USAGE_ENCRYPT
   - PSA_KEY_USAGE_DECRYPT
   - PSA_KEY_USAGE_SIGN_MESSAGE
   - PSA_KEY_USAGE_VERIFY_MESSAGE
   - PSA_KEY_USAGE_SIGN_HASH
   - PSA_KEY_USAGE_VERIFY_HASH
   - PSA_KEY_USAGE_DERIVE
   - PSA_KEY_USAGE_VERIFY_DERIVATION
   - PSA_KEY_USAGE_WRAP
   - PSA_KEY_USAGE_UNWRAP

   The flag PSA_KEY_USAGE_DERIVE_PUBLIC is used in the function psa_check_key_usage() to query if a key can be used for the public role in the specified algorithm.

psa_key_usage_t (typedef)

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 key-management usage flag permits a key to be moved outside of the security boundary of its current storage location. In particular:

- This flag is required to export a key from the cryptoprocessor using psa_export_key(). 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 make a copy of a key outside of a secure element using psa_copy_key(). See also PSA_KEY_USAGE_COPY.

If a key does not have export permission, implementations must not permit 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 key-management usage flag is required to make a copy of a key using psa_copy_key().

For a key lifetime that corresponds to a secure element location that enforces the non-exportability of keys, copying a key outside the secure element also requires the usage flag PSA_KEY_USAGE_EXPORT. Copying the key within 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 key-management usage flag permits 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 for the cryptoprocessor, to keep a copy of 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 42.

PSA_KEY_USAGE_ENCRYPT (macro)

Permission to encrypt a message, or perform key encapsulation, with the key.

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

This cryptographic-operation usage flag is required to use the key in a symmetric encryption operation, in an AEAD encryption-and-authentication operation, in an asymmetric encryption operation, or in a key-encapsulation operation. 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()
- psa_encapsulate()

For a key pair, this concerns the public key.

PSA_KEY_USAGE_DECRYPT (macro)

Permission to decrypt a message, or perform key decapsulation, with the key.

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

This cryptographic-operation usage flag is required to use the key in a symmetric decryption operation, in an AEAD decryption-and-verification operation, in an asymmetric decryption operation, or in a key-decapsulation operation. 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()
- psa_decapsulate()

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 cryptographic-operation usage flag is required to use the key in a MAC calculation operation, or in an asymmetric message signature operation. 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 cryptographic-operation usage flag is required to use the key in a MAC verification operation, or in an asymmetric message signature verification operation. 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 cryptographic-operation usage flag is required to use the key to sign a pre-computed message hash in an asymmetric signature operation. The flag must be present on keys used with the following APIs:

- 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 cryptographic-operation usage flag is required to use the key to verify a pre-computed message hash in an asymmetric signature verification operation. The flag must be present on keys used with the following APIs:

- 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 or produce a password hash from this key.

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

This cryptographic-operation usage flag is required to use the key for derivation in a key-derivation operation, or in a key-agreement operation.

This flag must be present on keys used with the following APIs:

- psa_key_agreement()

- psa_key_derivation_key_agreement()
- psa_raw_key_agreement()

If this flag is present on all keys used in calls to psa_key_derivation_input_key() for a key-derivation operation, then it permits calling psa_key_derivation_output_bytes(), psa_key_derivation_output_key(), psa_key_derivation_output_key_custom(), psa_key_derivation_verify_bytes(), or psa_key_derivation_verify_key() at the end of the operation.

PSA_KEY_USAGE_VERIFY_DERIVATION (macro)

Permission to verify the result of a key derivation, including password hashing.

Added in version 1.1.

```
#define PSA_KEY_USAGE_VERIFY_DERIVATION ((psa_key_usage_t)0x00008000)
```

This cryptographic-operation usage flag is required to use the key for verification in a key-derivation operation.

This flag must be present on keys used with psa_key_derivation_verify_key().

If this flag is present on all keys used in calls to psa_key_derivation_input_key() for a key-derivation operation, then it permits calling psa_key_derivation_verify_bytes() or psa_key_derivation_verify_key() at the end of the operation.

PSA_KEY_USAGE_DERIVE_PUBLIC (macro)

Used in the psa_check_key_usage() function to determine if the key can be used in the public key role in a key-agreement or a PAKE operation.

Added in version 1.4.

```
#define PSA_KEY_USAGE_DERIVE_PUBLIC ((psa_key_usage_t)0x00000080)
```

This cryptographic-operation usage flag is only used with the psa_check_key_usage() function. This flag is not currently checked when performing any cryptographic operation.

For example, calling psa_check_key_usage() with PSA_KEY_USAGE_DERIVE_PUBLIC and with:

- PSA_ALG_ECDH checks that the key can be used as the public share in the ECDH key agreement. There are no checks on permissions as the key share is provided in a buffer.
- PSA_ALG_SPAKE2P_HMAC will check that the key can be used in the Verifier role in the SPAKE2+ algorithm. The key must have the PSA_KEY_USAGE_DERIVE permission.
- PSA_ALG_HKDF is invalid, as there is no such role in single-key derivation algorithms.

PSA_KEY_USAGE_WRAP (macro)

Permission to wrap another key with the key.

```
#define PSA_KEY_USAGE_WRAP ((psa_key_usage_t)0x00010000)
```

This flag is required to use the key in a key-wrapping operation. The flag must be present on keys used with the following APIs:

- psa_wrap_key()

`PSA_KEY_USAGE_UNWRAP` (macro)

Permission to unwrap another key with the key.

```
#define PSA_KEY_USAGE_UNWRAP ((psa_key_usage_t)0x00020000)
```

This flag is required to use the key in a key-unwrapping operation. The flag must be present on keys used with the following APIs:

- `psa_unwrap_key()`

`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 100.

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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`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. It can be efficiently implemented as a `static inline` function or a function-like-macro.

`psa_check_key_usage` (function)

Query the capability of a key.

Added in version 1.4.

```
psa_status_t psa_check_key_usage(psa_key_id_t key,
                                 psa_algorithm_t alg,
                                 psa_key_usage_t usage);
```

Parameters

`key`	Identifier of the key to check.
`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.
`usage`	A single `PSA_KEY_USAGE_xxx` flag.

Returns: `psa_status_t`

`PSA_SUCCESS`	`key` can be used for the requested operation on this implementation.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `usage` is a key-management usage flag and `alg` is not `PSA_ALG_NONE`. • `usage` is a cryptographic-operation usage flag and `alg` is not a valid, specific algorithm. A 'specific algorithm' is one that is neither `PSA_ALG_NONE` nor a wildcard algorithm. • `usage` is not a valid role for algorithm `alg`. • `key` is not compatible with `alg` and `usage`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • The implementation does not support algorithm `alg`. • The implementation does not support using `key` with the operation associated with `alg` and `usage`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

This function reports whether the implementation supports the use of a key with the operation associated with a provided algorithm and usage. This function does not attempt to perform the operation.

If `usage` is a key-management usage flag, then:

- `alg` must be `PSA_ALG_NONE`.
- `key` must exist, and permit the requested usage flag.

If `usage` is a cryptographic-operation usage flag, then:

- `alg` must be a valid, fully specified algorithm, and not a wildcard. For example:
  - `PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)` is invalid as it is a wildcard algorithm.
  - `PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_GCM, 9)` is invalid as it has an invalid tag-length for GCM.
  - `PSA_ALG_SPAKE2P_HMAC(PSA_ALG_SHA_1)` is invalid as SPAKE2+ does have SHA-1 in any cipher-suite.
- `usage` must identify a valid role within the algorithm. For example, if `alg == PSA_ALG_GCM`, the `usage` must be either `PSA_KEY_USAGE_ENCRYPT` or `PSA_KEY_USAGE_DECRYPT`, as these are the key-usage policy flags for AEAD functions.
- `key` must exist, have a type and size that are compatible with the operation associated with `alg` and `usage`, and have the required permission for the algorithm and usage. For example:
  - An Edwards25519 key pair is not compatible with `PSA_ALG_ECDSA(PSA_ALG_SHA_256)`.
  - A `512`-bit RSA key pair is not compatible with `PSA_ALG_RSA_OAEP(PSA_ALG_SHA_512)` as the algorithm requires a larger key size.
  - A `512`-bit AES key (double-length key for use in AES-256-XTS) is not compatible with `PSA_ALG_CTR`.

Note:

For the key pair or public key of a valid type in a key agreement function, this function returns `PSA_SUCCESS` for the usage `PSA_KEY_USAGE_DERIVE_PUBLIC`, regardless of the key's policy. This is because the corresponding API functions take a key buffer as input, not a key object, and the key data can extracted by calling `psa_export_public_key()`, which does not require any usage flag.

Implementation note

The intended behavior of this function is to include any check that can be made using the accessible key attributes, but without requiring logic or arithmetic using the key material.

9.9 Key management functions

9.9.1 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()` and `psa_generate_key_custom()` create a key from randomly generated data.

- `psa_key_derivation_output_key()` and `psa_key_derivation_output_key_custom()` create a key from data generated by a pseudorandom derivation process. See *Key derivation* on page 244.
- `psa_key_agreement()` creates a key from the shared secret result of a key-agreement process. See *Key agreement* on page 317.
- `psa_encapsulate()` and `psa_decapsulate()` create a shared secret key using a key-encapsulation mechanism.
- `psa_pake_get_shared_key()` creates a key from the shared secret result of a password-authenticated key exchange. See *Password-authenticated key exchange (PAKE)* on page 338.
- `psa_copy_key()` duplicates an existing key with a different lifetime or with a more restrictive usage policy.
- `psa_attach_key()` registers implementation-provided key material for use as a volatile key.

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.

  The following attributes are required for all keys:

  - The key type determines how the `data` buffer is interpreted.

  The following attributes must be set for keys used in cryptographic operations:

  - The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
  - The key usage flags, see *Key usage flags* on page 102.

  The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

  - The key lifetime, see *Key lifetimes* on page 90.
  - The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

The following attributes are optional:

- If the key size is nonzero, it must be equal to the key size determined from `data`.

> **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 *Key format* section of the chosen key 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_BAD_STATE` — The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_NOT_PERMITTED` — The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.

`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 following conditions can result in this error:

- The key type is invalid.
- The key size is nonzero, and is incompatible with the key data in `data`.
- The key lifetime is invalid.
- The key identifier is not valid for the key lifetime.
- The key usage flags include invalid values.
- The key's permitted-usage algorithm is invalid.
- The key attributes, as a whole, are invalid.
- The key data is not correctly formatted for the key type.

`PSA_ERROR_NOT_SUPPORTED` — The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_INSUFFICIENT_STORAGE`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

The key is extracted from the provided `data` buffer. Its location, policy, and type are taken from `attributes`.

The provided 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` — the default value — 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 function supports any output from `psa_export_key()`. Each key type in *Key types* on page 53 describes the expected format of keys.

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 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_custom_key_parameters_t` (struct)

Custom production parameters for key generation or key derivation.

Added in version 1.3.

```
typedef struct psa_custom_key_parameters_t {
    uint32_t flags;
} psa_custom_key_parameters_t;
```

Fields

`flags`	Flags to control the key production process. `0` for the default production parameters.

Description

> **Note:**
>
> Future versions of the specification, and implementations, may add other fields in this structure.

The interpretation of this structure depends on the type of the key. Table 14 shows the custom production parameters for each type of key. See the key type definitions for details of the valid parameter values.

Table 14 Custom key parameters

Key type	Custom key parameters
RSA	Use the production parameters to select an exponent value that is different from the default value of `65537`. See `PSA_KEY_TYPE_RSA_KEY_PAIR`.
Other key types	Reserved for future use. `flags` must be `0`.

`PSA_CUSTOM_KEY_PARAMETERS_INIT` (macro)

The default production parameters for key generation or key derivation.

Added in version 1.3.

```
#define PSA_CUSTOM_KEY_PARAMETERS_INIT { 0 }
```

Calling `psa_generate_key_custom()` or `psa_key_derivation_output_key_custom()` with `custom == PSA_CUSTOM_KEY_PARAMETERS_INIT` and `custom_data_length == 0` is equivalent to calling `psa_generate_key()` or `psa_key_derivation_output_key()` respectively.

`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.

The following attributes are required for all keys:

- The key type. It must not be an asymmetric public key.
- The key size. It must be a valid size for the key type.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

> **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. For persistent keys, this is the key identifier defined in `attributes`. `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_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_NOT_PERMITTED`	The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
`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 following conditions can result in this error: • The key type is invalid, or is an asymmetric public-key type. • The key size is not valid for the key type. • The key lifetime is invalid. • The key identifier is not valid for the key lifetime. • The key usage flags include invalid values. • The key's permitted-usage algorithm is invalid. • The key attributes, as a whole, are invalid.
`PSA_ERROR_NOT_SUPPORTED`	The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 key type definitions in *Key types* on page 53 provide specific details relating to generation of the key.

> **Note:**
>
> This function is equivalent to calling `psa_generate_key_custom()` with the production parameters `PSA_CUSTOM_KEY_PARAMETERS_INIT` and `custom_data_length == 0` (`custom_data` is ignored).

`psa_generate_key_custom` (function)

Generate a key or key pair using custom production parameters.

Added in version 1.3.

```
psa_status_t psa_generate_key_custom(const psa_key_attributes_t * attributes,
                                     const psa_custom_key_parameters_t * custom,
                                     const uint8_t * custom_data,
                                     size_t custom_data_length,
                                     psa_key_id_t * key);
```

Parameters

`attributes`

The attributes for the new key.

The following attributes are required for all keys:

- The key type. It must not be an asymmetric public key.
- The key size. It must be a valid size for the key type.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

> **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.

`custom`

Customized production parameters for the key generation.

	When this is PSA_CUSTOM_KEY_PARAMETERS_INIT with custom_data_length == 0, this function is equivalent to psa_generate_key().
custom_data	A buffer containing additional variable-sized production parameters.
custom_data_length	Length of custom_data in bytes.
key	On success, an identifier for the newly created key. For persistent keys, this is the key identifier defined in attributes. 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_BAD_STATE	The library requires initializing by a call to psa_crypto_init().
PSA_ERROR_NOT_PERMITTED	The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
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 following conditions can result in this error: • The key type is invalid, or is an asymmetric public-key type. • The key size is not valid for the key type. • The key lifetime is invalid. • The key identifier is not valid for the key lifetime. • The key usage flags include invalid values. • The key's permitted-usage algorithm is invalid. • The key attributes, as a whole, are invalid. • The production parameters are invalid.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error: • The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location. • The production parameters are not supported by the implementation.
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

Description

Use this function to provide explicit production parameters when generating a key. See the description of `psa_generate_key()` for the operation of this function with the default production parameters.

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`.

See the documentation of `psa_custom_key_parameters_t` for a list of non-default production parameters. See the key type definitions in *Key types* on page 53 for details of the custom production parameters used for key generation.

`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 permit the usage `PSA_KEY_USAGE_COPY`. If a private or secret key is being copied outside of a secure element it must also permit `PSA_KEY_USAGE_EXPORT`.

`attributes`
: The attributes for the new key.

  The following attributes must be set for keys used in cryptographic operations:

  - The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
  - The key usage flags, see *Key usage flags* on page 102.

  These flags are combined with the source key policy so that both sets of restrictions apply, as described in the documentation of this function.

  The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

  - The key lifetime, see *Key lifetimes* on page 90.
  - The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

  The following attributes are optional:

  - If the key type has a non-default value, it must be equal to the source key type.
  - If the key size is nonzero, it must be equal to the source key size.

  > **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_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`source_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • `source_key` does not have the `PSA_KEY_USAGE_COPY` usage flag. • `source_key` does not have the `PSA_KEY_USAGE_EXPORT` usage flag, and the location of `target_key` is outside the security boundary of the `source_key` storage location. • The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
`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 following conditions can result in this error: • `attributes` specifies a key type or key size which does not match the attributes of `source key`. • The lifetime or identifier in `attributes` are invalid. • The key policies from `source_key` and those specified in `attributes` are incompatible.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • The `source key` storage location does not support copying to the target key's storage location. • The key attributes, as a whole, are not supported in the target key's storage location.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

Copy key material from one location to another. Its location is taken from `attributes`, its policy is the intersection of the policy in `attributes` and the source key policy, and its type and size are taken from the source key.

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 permits 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.

As a result, the new key cannot be used for operations that were not permitted on the source key.

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

psa_attach_key (function)

Register implementation-provided key material with a volatile key identifier and key policy.

Added in version 1.4.

```
psa_status_t psa_attach_key(const psa_key_attributes_t * attributes,
                            const uint8_t * label,
                            size_t label_length,
                            psa_key_id_t * key);
```

Parameters

attributes

The attributes for the key to be registered.

Some of the attributes can be provided by the implementation. It is IMPLEMENTATION DEFINED, and possibly key-specific, which attributes are provided by the implementation and which must be supplied by the application.

The following attributes must always be provided by the application:

- The key lifetime must specify a volatile key, and the storage location of the implementation-provided key. See *Key lifetimes* on page 90.

The following attributes must be provided by either the application or the implementation. If provided by both, they must be identical:

- The key type.
- The key size.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
- The key usage flags, see *Key usage flags* on page 102.

These attributes are combined with any policy that is provided by the implementation, so that both sets of restrictions apply.

`label` — Buffer containing a label that identifies the implementation-provided key to be registered.

The contents of this label are interpreted by the implementation and may correspond to a pre-provisioned, securely stored, or deterministically derived key within the location specified in `attributes`.

`label_length` — Size of the `label` 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.

`PSA_ERROR_BAD_STATE` — The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_NOT_PERMITTED` — The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.

`PSA_ERROR_INVALID_ARGUMENT` — The following conditions can result in this error:

- The key type is invalid.
- The key size is nonzero, and is incompatible with the implementation-provided key.
- The key lifetime specifies a non-volatile persistence level.
- The key lifetime specifies an invalid storage location.
- The key usage flags include invalid values.
- The key's permitted-usage algorithm is invalid.
- The key attributes, as a whole, are invalid.
- The implementation-provided key material is incompatible with the provided key attributes.

`PSA_ERROR_NOT_SUPPORTED` — The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location.

`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_DOES_NOT_EXIST` — `label` does not refer to key material within the location specified in `attributes`.

Description

This function allows applications to register implementation-provided key material. The key material can come from different sources, for example:

- Keys that are provisioned outside the Crypto API, such as during manufacturing or by a secure element.
- Keys that are deterministically derived from a secret within the implementation.

After registering the key, the application has a volatile key identifier that can be used in cryptographic operations permitted by its usage flags and algorithm policy.

The key material is identified by its location, specified in the provided attributes `lifetime` value, and the `label` parameter. The format of the label is specific to the implementation and storage location. Typically, the label is used as a location-specific identifier for the key material, or can provide input for deriving the key material from an internal secret.

This function can only be used to create a volatile key. That is, a key with a lifetime persistence level of `PSA_KEY_PERSISTENCE_VOLATILE`.

Depending on the key being registered, the implementation can provide some or all of the key type, size, and policy. For example:

- Provisioned key material has a fixed size. The implementation might permit the application to define the key type and policy, as long as these are compatible with the key material.
- An implementation-specific derived key can require the application to provide a key type and size, using these in the derivation process.
- An implementation-provided key can be fully defined by the implementation, with a fixed type, size, and policy. The call to `psa_attach_key()` needs to specify the location and label of the key, and a matching policy, in order to obtain a key id.

Calling `psa_destroy_key()` with a key identifier returned by `psa_attach_key()` will remove the key identifier and policy from the key store, but any implementation-provided key material remains within the implementation. A subsequent call to `psa_attach_key()` with the same parameters will return a new key identifier for the same key.

It is IMPLEMENTATION DEFINED whether the same implementation-provided key can be attached to multiple key identifiers concurrently.

Note:

This function is intended for scenarios where key material is provided outside the Crypto API, and the application needs to use such keys within the Crypto API framework.

The function only allows registering key material that is provided by the implementation. To import new key material, use `psa_import_key()`.

Although the implementation verifies that the application-supplied attributes are compatible with the implementation-provided key; it is the application's responsibility to ensure correctness for attributes that are provided by the implementation.

To create a persistent key from pre-existing key material, the implementation might permit a key returned by `psa_attach_key()` to be copied to a persistent key using `psa_copy_key()`.

Implementation note

Implementations may impose restrictions on which keys can be registered, depending on their storage architecture and security policies.

The behavior of a call `psa_attach_key()` with a persistent key-lifetime might be specified in a future version of the Crypto API. At present, it is recommended that such a call returns `PSA_ERROR_INVALID_ARGUMENT`, and does not provide implementation-specific behavior.

9.9.2 Key destruction

`psa_destroy_key` (function)

Destroy or unregister 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`	Success: • If `key` was a valid key identifier that was not the result of a call to `psa_attach_key()`, then material that it referred to has been erased. • If `key` was a valid key identifier that was returned by `psa_attach_key()`, then the key identifier is detached from the implementation-provided key. • Alternatively, `key` was `PSA_KEY_ID_NULL`.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is neither a valid key identifier, nor `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_COMMUNICATION_FAILURE`	There was an failure in communication with the cryptoprocessor. The key material might still be present in the cryptoprocessor.
`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_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	

Description

For key identifiers that resulted from registering an implementation-provided key using `psa_attach_key()`, this function detaches the key identifier from the implementation-provided key.

For other keys, 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	Success. The key material has been removed from memory, if the key material is not currently required.
PSA_ERROR_BAD_STATE	The library requires initializing by a call to `psa_crypto_init()`.
PSA_ERROR_INVALID_HANDLE	`key` is not a valid key identifier.
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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 42.

9.9.3 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 permit 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_ASYMMETRIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key or key pair. • 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`	Success. The first `(*data_length)` bytes of `data` contain the exported key.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The key does not have the `PSA_KEY_USAGE_EXPORT` flag.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `data` buffer is too small. `PSA_EXPORT_KEY_OUTPUT_SIZE()`, `PSA_EXPORT_KEY_PAIR_MAX_SIZE`, `PSA_EXPORT_PUBLIC_KEY_MAX_SIZE`, or `PSA_EXPORT_ASYMMETRIC_KEY_MAX_SIZE` can be used to determine a sufficient buffer size.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • The key's storage location does not support export of the key. • The implementation does not support export of keys with this key type.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 in *Key types* on page 53, not the originally imported representation.

For standard key types, the output format is defined in the relevant *Key format* section in *Key types* on page 53. 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.
- `PSA_EXPORT_ASYMMETRIC_KEY_MAX_SIZE` evaluates to the maximum output size of any supported public key or key pair.

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

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `(*data_length)` bytes of `data` contain the exported public key.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `data` buffer is too small. `PSA_EXPORT_PUBLIC_KEY_OUTPUT_SIZE()`, `PSA_EXPORT_PUBLIC_KEY_MAX_SIZE`, or `PSA_EXPORT_ASYMMETRIC_KEY_MAX_SIZE` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The key is neither a public key nor a key pair.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • The key's storage location does not support export of the key. • The implementation does not support export of keys with this key type.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 in *Key types* on page 53, not the originally imported representation.

For standard key types, the output format is defined in the relevant *Key format* section in *Key types* on page 53.

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

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`, `PSA_EXPORT_PUBLIC_KEY_MAX_SIZE`, and `PSA_EXPORT_ASYMMETRIC_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

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` and `PSA_EXPORT_ASYMMETRIC_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 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`, and `PSA_EXPORT_ASYMMETRIC_KEY_MAX_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 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()`, `PSA_EXPORT_KEY_OUTPUT_SIZE()`, `PSA_EXPORT_KEY_PAIR_MAX_SIZE`, and `PSA_EXPORT_ASYMMETRIC_KEY_MAX_SIZE`.

`PSA_EXPORT_ASYMMETRIC_KEY_MAX_SIZE` (macro)

Sufficient buffer size for exporting any asymmetric key pair or public key.

Added in version 1.3.

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

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

See also `PSA_EXPORT_KEY_PAIR_MAX_SIZE`, `PSA_EXPORT_PUBLIC_KEY_MAX_SIZE`, and `PSA_EXPORT_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 the Crypto API:

1. To specify a specific algorithm to use in a cryptographic operation. These are all defined in *Cryptographic operation reference*.
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 100.

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

- *Key-derivation algorithms* on page 245
- *Asymmetric signature* on page 278
- *Asymmetric encryption algorithms* on page 311
- *Key-agreement algorithms* on page 317
- *Key encapsulation* on page 329
- *Password-authenticated key exchange (PAKE)* on page 338

10.1.1 Algorithm encoding

`psa_algorithm_t` (typedef)

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`.

The *Algorithm and key type encoding* on page 410 appendix provides a full definition of the algorithm identifier encoding.

`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: a 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 138 for a list of defined hash algorithms.

PSA_ALG_IS_XOF (macro)

Whether the specified algorithm is an XOF algorithm.

Added in version 1.4.

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

Parameters

alg	An algorithm identifier: a value of type psa_algorithm_t.

Returns

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

Description

See *XOF algorithms* on page 158 for a list of defined XOF 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: a 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 165 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: a 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 182 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: a 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 208 for a list of defined AEAD algorithms.

`PSA_ALG_IS_KEY_WRAP` (macro)

Whether the specified algorithm is a key wrapping algorithm.

Added in version 1.4.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

Description

See *Key-wrapping algorithms* on page 237 for a list of defined key-wrapping 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: a 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 245 for a list of defined key-derivation 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: a 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* on page 278 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: a 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 311 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: a 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 317 for a list of defined key-agreement algorithms.

`PSA_ALG_IS_PAKE` (macro)

Whether the specified algorithm is a password-authenticated key exchange.

Added in version 1.1.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a password-authenticated key exchange (PAKE) algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported algorithm identifier.

`PSA_ALG_IS_KEY_ENCAPSULATION` (macro)

Whether the specified algorithm is a key-encapsulation algorithm.

Added in version 1.3.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

Description

See *Key encapsulation* on page 329 for a list of defined key-encapsulation algorithms.

10.1.3 Support macros

`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: a 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: a 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_DETERMINISTIC_ECDSA()
- PSA_ALG_ECDSA()
- PSA_ALG_HKDF()
- PSA_ALG_HKDF_EXPAND()
- PSA_ALG_HKDF_EXTRACT()
- PSA_ALG_HMAC()
- PSA_ALG_JPAKE()
- PSA_ALG_PBKDF2_HMAC()
- PSA_ALG_RSA_OAEP()
- PSA_ALG_RSA_PKCS1V15_SIGN()
- PSA_ALG_RSA_PSS()
- PSA_ALG_RSA_PSS_ANY_SALT()
- PSA_ALG_SP800_108_COUNTER_HMAC()
- PSA_ALG_SPAKE2P_CMAC()
- PSA_ALG_SPAKE2P_HMAC()
- PSA_ALG_TLS12_PRF()
- PSA_ALG_TLS12_PSK_TO_MS()

10.2 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. A multi-part hash operation is used as follows:

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()`.

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 applications.

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 applications.

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 applications.

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_AES_MMO_ZIGBEE` (macro)

The *Zigbee* 1.0 hash function based on a Matyas-Meyer-Oseas (MMO) construction using AES-128.

Added in version 1.2.

```
#define PSA_ALG_AES_MMO_ZIGBEE ((psa_algorithm_t)0x02000007)
```

This is the cryptographic hash function based on the Merkle-Damgård construction over a Matyas-Meyer-Oseas one-way compression function and the AES-128 block cipher, with the parametrization defined in *zigbee Specification* [ZIGBEE] §B.6.

This hash function can operate on input strings of up to $2^{32} - 1$ bits.

> **Note:**
>
> The Zigbee keyed hash function from [ZIGBEE] §B.1.4 is `PSA_ALG_HMAC(PSA_ALG_AES_MMO_ZIGBEE)`.

`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 applications.

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_SHAKE256_512` (macro)

The first 512 bits (64 bytes) of the output from SHAKE256.

Added in version 1.1.

```
#define PSA_ALG_SHAKE256_512 ((psa_algorithm_t)0x02000015)
```

This is used for pre-hashing in Ed448ph, see `PSA_ALG_ED448PH`.

The SHAKE256 XOF is defined in [FIPS202].

> **Note:**
>
> To use SHAKE256 as an XOF, see *Extendable-output functions (XOF)* on page 157 and `PSA_ALG_SHAKE256`.

> **Note:**
>
> For other scenarios where a hash function based on SHA3 or SHAKE is required, SHA3-512 is recommended. SHA3-512 has the same output size, and a theoretically higher security strength.

`PSA_ALG_SM3` (macro)

The SM3 message-digest algorithm.

```
#define PSA_ALG_SM3 ((psa_algorithm_t)0x02000014)
```

SM3 is defined in *ISO/IEC 10118-3:2018 IT Security techniques — Hash-functions — Part 3: Dedicated hash-functions* [ISO10118], and also in *GM/T 0004-2012: SM3 cryptographic hash algorithm* [CSTC0004].

`PSA_ALG_ASCON_HASH256` (macro)

The Ascon-Hash256 message-digest algorithm.

Added in version 1.4.

```
#define PSA_ALG_ASCON_HASH256 ((psa_algorithm_t)0x02000019)
```

Ascon-Hash256 is defined in *NIST Special Publication 800-232: Ascon-Based Lightweight Cryptography Standards for Constrained Devices* [SP800-232] §5.1.

> **Note:**
>
> To use the Ascon XOF algorithms, see `PSA_ALG_ASCON_XOF128` and `PSA_ALG_ASCON_CXOF128`.

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: a value of type `psa_algorithm_t` 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. The first `(*hash_length)` bytes of `hash` contain the hash value.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `hash` buffer is too small. `PSA_HASH_LENGTH()` can be used to determine a sufficient buffer size.

PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error: • alg is not a hash algorithm. • input_length is too large for alg.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error: • alg is not supported or is not a hash algorithm. • input_length is too large for the implementation.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	

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: a value of type psa_algorithm_t 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	Success. The expected hash is identical to the actual hash of the input.
PSA_ERROR_BAD_STATE	The library requires initializing by a call to psa_crypto_init().
PSA_ERROR_INVALID_SIGNATURE	The calculated hash of the message does not match the value in hash.
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error: • alg is not a hash algorithm. • input_length is too large for alg.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:

- `alg` is not supported or is not a hash algorithm.
- `input_length` is too large for the implementation.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

10.2.3 Multi-part hashing operations

`psa_hash_operation_t` (typedef)

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 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: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`alg` is not a hash algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a hash algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

The sequence of operations to calculate a hash (message digest) is as follows:

1. Allocate a hash 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()`.

After a successful call to `psa_hash_setup()`, the operation is active, and 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()`.

If `psa_hash_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_hash_abort()`.

See *Multi-part operations* on page 27.

`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 following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The total input for the operation is too large for the hash algorithm.
`PSA_ERROR_NOT_SUPPORTED`	The total input for the operation is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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. The first `(*hash_length)` bytes of `hash` contain the hash value.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `hash` buffer is too small. `PSA_HASH_LENGTH()` can be used to determine a sufficient buffer size.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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`	Success. The expected hash is identical to the actual hash of the message.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_SIGNATURE`	The calculated hash of the message does not match the value in `hash`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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. The first `(*hash_state_length)` bytes of `hash_state` contain the intermediate hash state.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.

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 a sufficient 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_CORRUPTION_DETECTED	

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 155 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 155.
hash_state_length	Length of hash_state in bytes.

Returns: psa_status_t

PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to psa_crypto_init().
PSA_ERROR_INVALID_ARGUMENT	hash_state does not correspond to a valid hash suspend state. See *Hash suspend state format* on page 155 for the definition.
PSA_ERROR_NOT_SUPPORTED	The provided hash suspend state is for an algorithm that is not supported.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	

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`	Success. `target_operation` is ready to continue the same hash operation as `source_operation`.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The `source_operation` state is not valid: it must be active. • The `target_operation` state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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 or an HMAC algorithm: a value of type `psa_algorithm_t` such that `(PSA_ALG_IS_HASH(alg) || PSA_ALG_IS_HMAC(alg))` is true.

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 */
```

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()`, in bytes.

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

Parameters

`alg`	A hash algorithm: a value of type `psa_algorithm_t` 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 */
```

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.

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: a value of type `psa_algorithm_t` 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 157.

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: a value of type `psa_algorithm_t` 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 157.

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: a value of type `psa_algorithm_t` 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 SM3 algorithm and 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 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 *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-512/224, SHA-512/256, SHA-384, and SHA-512: *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 157.

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-512/224, SHA-512/256, SHA-384, and SHA-512: 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* on page 157.

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:

$$\text{length}(unprocessed\_input) = input\_length \mod hash\_block\_size.$$

The value of *hash_block_size* is specific to the hash 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* on page 157.

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(alg)` 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_224`	16	64	0 – 127
`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 Extendable-output functions (XOF)

An eXtendable-Output Function (XOF) is similar to a cryptographic hash, transforming an arbitrary amount of input data into pseudorandom output. Unlike hash algorithms, an XOF can produce an arbitrary amount of output.

XOF algorithms are often used as a building block in other algorithms, as they are suitable for use in hashing, key-derivation, and as a pseudorandom function (PRF).

In the Crypto API, support for XOF algorithms is provided by the `psa_xof_operation_t` multi-part operation, and XOF algorithm identifiers. A multi-part XOF operation is used as follows:

1. Initialize the `psa_xof_operation_t` object to zero, or by assigning the value of the associated macro `PSA_XOF_OPERATION_INIT`.
2. Call `psa_xof_setup()` to specify the required XOF algorithm.
3. If the algorithm has a context, call `psa_xof_set_context()` to provide the context value.
4. Call the `psa_xof_update()` function on successive chunks of the input data.

5. After input is complete, call `psa_xof_output()` one or more times to extract successive chunks of output.
6. When output is complete, call `psa_xof_abort()` to end the operation.

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

Note:

For an XOF algorithm:

- The result does not depend on how the overall input is fragmented. For example, calling `psa_xof_update()` twice with input i_1 and i_2 has the same effect as calling `psa_xof_update()` once with the concatenation $i_1 \mid\mid i_2$.
- The overall output does not depend on how the output is fragmented. If the output is considered as a stream of bytes, `psa_xof_output()` is an operation that reads bytes in sequence from the stream of data.

10.3.1 XOF algorithms

`PSA_ALG_SHAKE128` (macro)

The SHAKE128 XOF algorithm.

Added in version 1.4.

```
#define PSA_ALG_SHAKE128 ((psa_algorithm_t)0x0D000100)
```

SHAKE128 is one of the KECCAK family of algorithms.

SHAKE128 is defined in *FIPS Publication 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* [FIPS202].

Some fixed output-length hash algorithms based on SHAKE128 are also provided in the Crypto API:

- `PSA_ALG_SHAKE128_256` — defined in *PSA Certified Crypto API 1.4 PQC Extension* [PSA-PQC]

`PSA_ALG_SHAKE256` (macro)

The SHAKE256 XOF algorithm.

Added in version 1.4.

```
#define PSA_ALG_SHAKE256 ((psa_algorithm_t)0x0D000200)
```

SHAKE256 is one of the KECCAK family of algorithms.

SHAKE256 is defined in [FIPS202].

Some fixed output-length hash algorithms based on SHAKE256 are also provided in the Crypto API:

- `PSA_ALG_SHAKE256_192` — defined in [PSA-PQC]
- `PSA_ALG_SHAKE256_256` — defined in [PSA-PQC]
- `PSA_ALG_SHAKE256_512`

`PSA_ALG_ASCON_XOF128` (macro)

The Ascon-XOF128 XOF algorithm.

Added in version 1.4.

```
#define PSA_ALG_ASCON_XOF128 ((psa_algorithm_t)0x0D000300)
```

Ascon-XOF128 is defined in *NIST Special Publication 800-232: Ascon-Based Lightweight Cryptography Standards for Constrained Devices* [SP800-232] §5.2.

> **Note:**
>
> To use the Ascon-Hash256 hash algorithm, see `PSA_ALG_ASCON_HASH256`.

`PSA_ALG_ASCON_CXOF128` (macro)

The Ascon-CXOF128 XOF algorithm, with context.

Added in version 1.4.

```
#define PSA_ALG_ASCON_CXOF128 ((psa_algorithm_t)0x0D008300)
```

Ascon-CXOF128 is defined in *NIST Special Publication 800-232: Ascon-Based Lightweight Cryptography Standards for Constrained Devices* [SP800-232] §5.3.

The context value must be provided by calling `psa_xof_set_context()` on the XOF mluti-part operation, before providing any input data.

10.3.2 Multi-part XOF operations

`psa_xof_operation_t` (typedef)

The type of the state object for multi-part XOF operations.

Added in version 1.4.

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

Before calling any function on an XOF operation object, the application must initialize it by any of the following means:

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

  ```
  psa_xof_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_xof_operation_t operation;
  ```

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

```
psa_xof_operation_t operation = PSA_XOF_OPERATION_INIT;
```

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

```
psa_xof_operation_t operation;
operation = psa_xof_operation_init();
```

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

`PSA_XOF_OPERATION_INIT` (macro)

This macro returns a suitable initializer for an XOF operation object of type `psa_xof_operation_t`.

Added in version 1.4.

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

`psa_xof_operation_init` (function)

Return an initial value for an XOF operation object.

Added in version 1.4.

```
psa_xof_operation_t psa_xof_operation_init(void);
```

Returns: `psa_xof_operation_t`

`psa_xof_setup` (function)

Set up an XOF operation.

Added in version 1.4.

```
psa_status_t psa_xof_setup(psa_xof_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_xof_operation_t` and not yet in use.
`alg`	The XOF algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_XOF(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`alg` is not an XOF algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not an XOF algorithm.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

Description

The sequence of operations to generate XOF output is as follows:

1. Allocate an XOF 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_xof_operation_t`, e.g. `PSA_XOF_OPERATION_INIT`.
3. Call `psa_xof_setup()` to specify the algorithm.
4. For an XOF algorithm that has a context, call `psa_xof_set_context()` to provide the context.
5. Call `psa_xof_update()` zero, one, or more times, passing a fragment of the input each time.
6. To extract XOF output data, call `psa_xof_output()` one or more times.

After a successful call to `psa_xof_setup()`, the operation is active, and the application must eventually terminate the operation with a call to `psa_xof_abort()`.

If `psa_xof_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_xof_abort()`.

See *Multi-part operations* on page 27.

`psa_xof_set_context` (function)

Provide a context for a multi-part XOF operation.

Added in version 1.4.

```
psa_status_t psa_xof_set_context(psa_xof_operation_t * operation,
                                 const uint8_t * context,
                                 size_t context_length);
```

Parameters

`operation`	Active XOF operation.
`context`	Buffer containing the input fragment.
`context_length`	Size of the `context` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and no call to `psa_xof_set_context()`, `psa_xof_output()`, or `psa_xof_output()` has been made. • The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The algorithm does not support a context value. See `PSA_ALG_XOF_HAS_CONTEXT()`. • The context value is not valid for the XOF algorithm.
`PSA_ERROR_NOT_SUPPORTED`	The context value is not supported by this implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

This function sets the context value in a multi-part XOF operation, when using an XOF algorithm that has a context parameter.

The application must call `psa_xof_setup()` before calling this function. For an XOF algorithm with a context parameter, this function must be called immediately after `psa_xof_setup()`, before calling any other function on the XOF operation.

This function must not be called if the XOF algorithm does not have a context parameter. The macro `PSA_ALG_XOF_HAS_CONTEXT()` can be used to determine if a context value is required for the XOF algorithm.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_xof_abort()`.

`psa_xof_update` (function)

Add input to a multi-part XOF operation.

Added in version 1.4.

```
psa_status_t psa_xof_update(psa_xof_operation_t * operation,
                            const uint8_t * input,
                            size_t input_length);
```

Parameters

`operation`	Active XOF operation.
`input`	Buffer containing the input fragment.
`input_length`	Size of the `input` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and no call to `psa_xof_output()` has been made. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The total input for the operation is too large for the XOF algorithm.
`PSA_ERROR_NOT_SUPPORTED`	The total input for the operation is too large for the implementation.

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

Description

The application must call `psa_xof_setup()` before calling this function.

This function can be called zero, one, or more times to provide input for the XOF. The input to the XOF is only finalized on the first call to `psa_xof_output()`.

`psa_xof_update()` cannot be called on an XOF operation once `psa_xof_output()` has been called on the operation.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_xof_abort()`.

`psa_xof_output` (function)

Extract data from an XOF operation.

Added in version 1.4.

```
psa_status_t psa_xof_output(psa_xof_operation_t * operation,
                            uint8_t * output,
                            size_t output_length);
```

Parameters

`operation`	Active XOF operation.
`output`	Buffer where the output will be written.
`output_length`	Number of bytes to output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `output_length` bytes of `output` contain the data.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

This function calculates output bytes from the XOF 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 application must call `psa_xof_setup()` and supply all input data, using calls to `psa_xof_update()`, before calling this function. The input to the XOF is finalized on the first call to `psa_xof_output()` before data is

extracted from the XOF.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_xof_abort()`.

`psa_xof_abort` (function)

Abort an XOF operation.

Added in version 1.4.

```
psa_status_t psa_xof_abort(psa_xof_operation_t * operation);
```

Parameters

`operation`	Initialized XOF operation.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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_xof_setup()` again.

This function can be called any time after the operation object has been initialized by one of the methods described in `psa_xof_operation_t`.

In particular, calling `psa_xof_abort()` after the operation has been terminated by a call to `psa_xof_abort()` is safe and has no effect.

10.3.3 Support macros

`PSA_ALG_XOF_HAS_CONTEXT` (macro)

Whether the specified XOF algorithm has a context parameter.

Added in version 1.4.

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

Parameters

`alg`	An XOF algorithm identifier: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_XOF(alg)` is true.

Returns

`1` if `alg` is an XOF algorithm that has a context parameter. `0` if `alg` is an XOF algorithm that does not have a context parameter. This macro can return either `0` or `1` if `alg` is not a supported XOF algorithm identifier.

10.4 Message authentication codes (MAC)

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. A multi-part MAC operation is used as follows:

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()`.

10.4.1 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: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. See below on selecting a hash algorithm for use with HMAC.

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].

Choice of hash algorithm

An HMAC block size must be defined for use with each hash algorithm, which is at least as large as the hash output size.

HMAC was designed for hashes that use a Merkle-Damgård construction, for example, MD5, SHA-1, and SHA-2. For these hash algorithms, the HMAC block size is defined to be the hash input-block size.

Some algorithms do not have a defined HMAC block size. For example, Ascon (PSA_ALG_ASCON_HASH256) or Shake-based hashes (PSA_ALG_SHAKE256_512).

Table 15 lists the valid hash algorithms for use with HMAC, and their HMAC block and output sizes in bytes.

Table 15 Hash algorithms that can be used with HMAC

Algorithm	HMAC block size	Output size
PSA_ALG_MD2	16	16
PSA_ALG_MD4	64	16
PSA_ALG_MD5	64	16
PSA_ALG_RIPEMD160	64	20
PSA_ALG_SHA_1	64	20
PSA_ALG_SHA_224	64	28
PSA_ALG_SHA_256	64	32
PSA_ALG_SHA_384	128	48
PSA_ALG_SHA_512	128	64
PSA_ALG_SHA_512_224	128	28
PSA_ALG_SHA_512_256	128	32
PSA_ALG_SHA3_224	144	28
PSA_ALG_SHA3_256	136	32
PSA_ALG_SHA3_384	104	48
PSA_ALG_SHA3_512	72	64
PSA_ALG_SM3	64	32

Implementation note

It is recommended that other hash algorithms are not supported with PSA_ALG_HMAC. Future versions of the Crypto API might specify HMAC support for these hash algorithms, and will define the block size to use for HMAC.

Compatible key types

PSA_KEY_TYPE_HMAC

`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].

Compatible key types

`PSA_KEY_TYPE_AES`

`PSA_KEY_TYPE_ARIA`

`PSA_KEY_TYPE_DES`

`PSA_KEY_TYPE_CAMELLIA`

`PSA_KEY_TYPE_SM4`

`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].

Compatible key types

`PSA_KEY_TYPE_AES`

`PSA_KEY_TYPE_ARIA`

`PSA_KEY_TYPE_DES`

`PSA_KEY_TYPE_CAMELLIA`

`PSA_KEY_TYPE_SM4`

`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: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_MAC(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 untruncated 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 `mac_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 untruncated MAC algorithm can be recovered using `PSA_ALG_FULL_LENGTH_MAC()`.

Compatible key types

The resulting truncated MAC algorithm is compatible with the same key types as the MAC algorithm used to construct it.

`PSA_ALG_FULL_LENGTH_MAC` (macro)

Macro to construct the MAC algorithm with an untruncated MAC, from a truncated MAC algorithm.

```
#define PSA_ALG_FULL_LENGTH_MAC(mac_alg) /* specification-defined value */
```

Parameters

`mac_alg`	A MAC algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_MAC(mac_alg)` is true. This can be a truncated or untruncated MAC algorithm.

Returns

The corresponding MAC algorithm with an untruncated MAC.

Unspecified if `mac_alg` is not a supported MAC algorithm.

Compatible key types

The resulting untruncated MAC algorithm is compatible with the same key types as the MAC algorithm used to construct it.

`PSA_ALG_AT_LEAST_THIS_LENGTH_MAC` (macro)

Macro to build a MAC minimum-MAC-length wildcard algorithm.

Added in version 1.1.

```
#define PSA_ALG_AT_LEAST_THIS_LENGTH_MAC(mac_alg, min_mac_length) \
    /* specification-defined value */
```

Parameters

`mac_alg`	A MAC algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_MAC(alg)` is true. This can be a truncated or untruncated MAC algorithm.
`min_mac_length`	Desired minimum length of the message authentication code in bytes. This must be at most the untruncated length of the MAC and must be at least `1`.

Returns

The corresponding MAC wildcard algorithm with the specified minimum MAC length.

Unspecified if `mac_alg` is not a supported MAC algorithm or if `min_mac_length` is less than `1` or too large for the specified MAC algorithm.

Description

A key with a minimum-MAC-length MAC wildcard algorithm as permitted-algorithm policy can be used with all MAC algorithms sharing the same base algorithm, and where the (potentially truncated) MAC length of the specific algorithm is equal to or larger then the wildcard algorithm's minimum MAC length.

> **Note:**
>
> When setting the minimum required MAC length to less than the smallest MAC length permitted by the base algorithm, this effectively becomes an 'any-MAC-length-permitted' policy for that base algorithm.

The untruncated MAC algorithm can be recovered using `PSA_ALG_FULL_LENGTH_MAC()`.

Compatible key types

The resulting wildcard MAC algorithm is compatible with the same key types as the MAC algorithm used to construct it.

10.4.2 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 permit the usage `PSA_KEY_USAGE_SIGN_MESSAGE`.
`alg`	The MAC algorithm to compute: a value of type `psa_algorithm_t` 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. The first `(*mac_length)` bytes of `mac` contain the MAC value.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 `mac` buffer is too small. `PSA_MAC_LENGTH()` or `PSA_MAC_MAX_SIZE` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error:

- `alg` is not a MAC algorithm.
- `key` is not compatible with `alg`.
- `input_length` is too large for `alg`.

`PSA_ERROR_NOT_SUPPORTED` The following conditions can result in this error:

- `alg` is not supported or is not a MAC algorithm.
- `key` is not supported for use with `alg`.
- `input_length` is too large for the implementation.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

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 permit the usage `PSA_KEY_USAGE_VERIFY_MESSAGE`.

`alg` The MAC algorithm to compute: a value of type `psa_algorithm_t` 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`	Success. The expected MAC is identical to the actual MAC of the input.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 calculated MAC of the message does not match the value in `mac`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not a MAC algorithm. • `key` is not compatible with `alg`. • `input_length` is too large for `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not a MAC algorithm. • `key` is not supported for use with `alg`. • `input_length` is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

10.4.3 Multi-part MAC operations

`psa_mac_operation_t` (typedef)

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 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 permit the usage `PSA_KEY_USAGE_SIGN_MESSAGE`.
`alg`	The MAC algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_MAC(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive.

	• The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	The following conditions can result in this error: • `alg` is not a MAC algorithm. • `key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not a MAC algorithm. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 a MAC 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.

After a successful call to `psa_mac_sign_setup()`, the operation is active, and the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to `psa_mac_sign_finish()`.
- A call to `psa_mac_abort()`.

If `psa_mac_sign_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_mac_abort()`.

See *Multi-part operations* on page 27.

`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 permit the usage `PSA_KEY_USAGE_VERIFY_MESSAGE`.
`alg`	The MAC algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_MAC(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	The following conditions can result in this error: • `alg` is not a MAC algorithm. • `key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not a MAC algorithm. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 a MAC 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.

After a successful call to `psa_mac_verify_setup()`, the operation is active, and the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to `psa_mac_verify_finish()`.
- A call to `psa_mac_abort()`.

If `psa_mac_verify_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_mac_abort()`.

See *Multi-part operations* on page 27.

`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 following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The total input for the operation is too large for the MAC algorithm.

PSA_ERROR_NOT_SUPPORTED	The total input for the operation is too large for the implementation.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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_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.

Returns: `psa_status_t`

PSA_SUCCESS	Success. The first `(*mac_length)` bytes of `mac` contain the MAC value.
PSA_ERROR_BAD_STATE	The following conditions can result in this error: • The operation state is not valid: it must be an active mac sign operation. • The library requires initializing by a call to `psa_crypto_init()`.

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 a sufficient buffer size.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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	Success. The expected MAC is identical to the actual MAC of the message.
PSA_ERROR_BAD_STATE	The following conditions can result in this error: • The operation state is not valid: it must be an active mac verify operation. • The library requires initializing by a call to psa_crypto_init().

`PSA_ERROR_INVALID_SIGNATURE`	The calculated MAC of the message does not match the value in `mac`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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.

10.4.4 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: a 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: a 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_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: a value of type psa_algorithm_t 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

If the size of the MAC buffer is at least this large, it is guaranteed that `psa_mac_compute()` and `psa_mac_sign_finish()` will not fail due to an insufficient buffer size.

This is also the MAC length that `psa_mac_verify()` and `psa_mac_verify_finish()` expect.

See also `PSA_MAC_MAX_SIZE`.

`PSA_MAC_MAX_SIZE` (macro)

A sufficient buffer size for storing the MAC output by `psa_mac_verify()` and `psa_mac_verify_finish()`, for any of the supported key types and MAC algorithms.

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

If the size of the MAC buffer is at least this large, it is guaranteed that `psa_mac_verify()` and `psa_mac_verify_finish()` will not fail due to an insufficient buffer size.

See also `PSA_MAC_LENGTH()`.

10.5 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 207.

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 initialization vector (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. A multi-part cipher operation is used as follows:

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 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().

10.5.1 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, ChaCha20, and XChaCha20 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 applications.

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* [RFC8439] §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 [RFC8439] 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 are 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.

XChaCha20

To use XChaCha20, use a key type of `PSA_KEY_TYPE_XCHACHA20` and algorithm id `PSA_ALG_STREAM_CIPHER`.

XChaCha20 is a variation of ChaCha20 that uses a 192-bit nonce and a 64-bit counter. The larger nonce provides much lower probability of nonce misuse.

When using an XChaCha20 key with the `PSA_ALG_STREAM_CIPHER` algorithm, the nonce and an initial counter values are provided using the initialization vector (IV) functions in the following ways:

- A call to `psa_cipher_encrypt()` will generate a random 24-byte nonce, and set the counter value to zero. The random nonce is output as a 24-byte IV value in the output.
- A call to `psa_cipher_decrypt()` will use first 24 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 24-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:
  - 24 bytes: the provided IV is used as the nonce, and the counter value is set to zero.
  - 32 bytes: the first 8 bytes of the IV are used as the counter value (encoded as little-endian), and the remaining 24 bytes are used as the nonce.

  Other sizes of IV are invalid.

XChaCha20 is defined in *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305* [XCHACHA].

Compatible key types

`PSA_KEY_TYPE_ARC4`

`PSA_KEY_TYPE_CHACHA20`

`PSA_KEY_TYPE_XCHACHA20`

`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 operates using 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.

For the `PSA_ALG_CTR` algorithm, the counter block is initialized from the IV. The counter block is then treated as a single, big-endian encoded integer, and the counter block is updated by incrementing this integer by `1`.

The security of CTR mode depends on using counter block values that are unique across all messages encrypted using the same key value. This is achieved by using suitable initial counter block values, the appropriate way to do this depends on the application use case:

- If the application is using CTR mode to implement a protocol that specifies the construction of the IV, then the application must use a multi-part cipher operation, and call `psa_cipher_set_iv()` with the appropriate IV for encryption and decryption operations.

  > **Note:**
  > The protocol must use the same counter block update strategy as the one specified here.

- If the application is able to construct a unique *nonce* value for each time the same key is used to encrypt data, then it is recommended that the application uses a multi-part cipher operation, and call `psa_cipher_set_iv()` using the nonce as the IV for encryption and decryption operations.

  The nonce length, n bytes, must satisfy $1 \leq n \leq b$, where b is the cipher block size in bytes. To avoid a counter-block collision with other nonce values, the application must ensure that at most $2^{8(b-n)}$ blocks of data are encrypted in any single operation.

  For example, when using CTR encryption with an AES key, the cipher block size is 16 bytes. 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.

- Otherwise, it is recommended that the application uses a random IV value when encrypting data, and transmits the IV along with the ciphertext for use when decrypting the data. This can be achieved with either the single-part cipher functions or the multi-part cipher operation:
  - In a multi-part cipher encryption operation, call `psa_cipher_generate_iv()`, which returns the IV value. To use the same IV in a multi-part cipher decryption operation, call `psa_cipher_set_iv()`.
  - 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.

When using `PSA_ALG_CTR`, if the IV passed to `psa_cipher_set_iv()` is shorter than a cipher block, the initial counter block is formed by padding the end of the IV with zero bytes up to the block length.

> **Note:**
> The cipher block length can be determined using `PSA_BLOCK_CIPHER_BLOCK_LENGTH()`.

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_DES

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

PSA_ALG_CCM_STAR_NO_TAG (macro)

The CCM* cipher mode without authentication.

Added in version 1.2.

```
#define PSA_ALG_CCM_STAR_NO_TAG ((psa_algorithm_t)0x04c01300)
```

This is CCM* as specified in *IEEE Standard for Low-Rate Wireless Networks* [IEEE-CCM] §7, with a tag length of 0. For CCM* with a nonzero tag length, use the AEAD algorithm PSA_ALG_CCM.

The underlying block cipher is determined by the key type.

The IV generated or set in the cipher API is used as the nonce in the CCM* operation. An implementation must support the default IV length of 13. Support for setting a shorter IV is optional.

The maximum message length that can be encrypted is dependent on the length of the IV. See PSA_ALG_CCM for details of this relationship.

Usage in Zigbee

The Zigbee message encryption algorithm is based on CCM*. This is detailed in *zigbee Specification* [ZIGBEE] §B.1.1 and §A.

- For unauthenticated messages — when tag length $M = 0$ — the PSA_ALG_CCM_STAR_NO_TAG algorithm is used with an AES-128 key in a multi-part cipher operation. The 13-byte IV must be constructed as specified in [ZIGBEE], and provided to the operation using psa_cipher_set_iv().

  > **Note:**
  > An implementation of Zigbee cannot use the single-part psa_cipher_encrypt() function, as this generates a random IV, which is not valid for the Zigbee protocol.

- For authenticated messages — when tag length $M \in \{4, 8, 16\}$ — the PSA_ALG_AEAD_WITH_SHORTENED_TAG(PSA_ALG_CCM, tag_length) algorithm is used with an AES-128 key, where tag_length is the required value of M. The 13-byte nonce must be constructed as specified in [ZIGBEE].

  As the default tag length for CCM is 16, then PSA_ALG_CCM algorithm can be used when $M = 16$.
- To enable a single AES-128 key to be used for both the PSA_ALG_CCM_STAR_NO_TAG cipher and PSA_ALG_CCM AEAD algorithm, the key can be defined with the wildcard PSA_ALG_CCM_STAR_ANY_TAG permitted algorithm.

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

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.

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_DES

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

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].

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_DES

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

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].

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_DES

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

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].

Compatible key types

`PSA_KEY_TYPE_AES`

`PSA_KEY_TYPE_ARIA`

`PSA_KEY_TYPE_DES`

`PSA_KEY_TYPE_CAMELLIA`

`PSA_KEY_TYPE_SM4`

`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].

Compatible key types

`PSA_KEY_TYPE_AES`

`PSA_KEY_TYPE_ARIA`

`PSA_KEY_TYPE_DES`

`PSA_KEY_TYPE_CAMELLIA`

`PSA_KEY_TYPE_SM4`

`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.

Compatible key types

`PSA_KEY_TYPE_AES`

`PSA_KEY_TYPE_ARIA`

`PSA_KEY_TYPE_DES`

`PSA_KEY_TYPE_CAMELLIA`

`PSA_KEY_TYPE_SM4`

10.5.2 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 permit the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The cipher algorithm to compute: a value of type `psa_algorithm_t` 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. The first `(*output_length)` bytes of `output` contain the encrypted output.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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_CIPHER_ENCRYPT_OUTPUT_SIZE()` or `PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE()` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not a cipher algorithm. • `key` is not compatible with `alg`. • 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`	The following conditions can result in this error: • `alg` is not supported or is not a cipher algorithm. • `key` is not supported for use with `alg`. • `input_length` is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 permit the usage `PSA_KEY_USAGE_DECRYPT`.
`alg`	The cipher algorithm to compute: a value of type `psa_algorithm_t` 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. The first `(*output_length)` bytes of `output` contain the plaintext.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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_CIPHER_DECRYPT_OUTPUT_SIZE() or PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE() can be used to determine a sufficient buffer size.
PSA_ERROR_INVALID_PADDING	The algorithm uses padding, and the input does not contain valid padding.
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error: • alg is not a cipher algorithm. • key is not compatible with alg. • 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	The following conditions can result in this error: • alg is not supported or is not a cipher algorithm. • key is not supported for use with alg. • input_length is too large for the implementation.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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.

10.5.3 Multi-part cipher operations

psa_cipher_operation_t (typedef)

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 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 permit the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The cipher algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_CIPHER(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	The following conditions can result in this error: • `alg` is not a cipher algorithm. • `key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not a cipher algorithm. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

The sequence of operations to encrypt a message with a symmetric cipher is as follows:

1. Allocate a cipher 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()`.

After a successful call to `psa_cipher_encrypt_setup()`, the operation is active, and 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()`.

If `psa_cipher_encrypt_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_cipher_abort()`.

See *Multi-part operations* on page 27.

`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 permit the usage `PSA_KEY_USAGE_DECRYPT`.
`alg`	The cipher algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_CIPHER(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	The following conditions can result in this error: • `alg` is not a cipher algorithm. • `key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not a cipher algorithm. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

The sequence of operations to decrypt a message with a symmetric cipher is as follows:

1. Allocate a cipher 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()`.

After a successful call to `psa_cipher_decrypt_setup()`, the operation is active, and 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()`.

If `psa_cipher_decrypt_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_cipher_abort()`.

See *Multi-part operations* on page 27.

`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. The first `(*iv_length)` bytes of `iv` contain the generated IV.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The cipher algorithm does not use an IV.

- The operation state is not valid: it must be active, with no IV set.
- The library requires initializing by a call to `psa_crypto_init()`.

`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 a sufficient buffer size.

`PSA_ERROR_INSUFFICIENT_ENTROPY`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

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` — The following conditions can result in this error:

- 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. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The chosen algorithm does not use an IV. • `iv_length` is not valid for the chosen algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`iv_length` is not supported for use with the operation's algorithm and key.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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. The first `(*output_length)` bytes of `output` contain the output data.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, with an IV set if required for the algorithm. • The library requires initializing by a call to `psa_crypto_init()`.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The total input size passed to this operation is too large for this particular algorithm.
`PSA_ERROR_NOT_SUPPORTED`	The total input size passed to this operation is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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()`.

> **Note:**
>
> 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_cipher_update()` provides sufficient input, or a subsequent call to `psa_cipher_finish()` indicates the end of the input. The amount of data that can be delayed in this way is bounded by the associated output size macro: `PSA_CIPHER_UPDATE_OUTPUT_SIZE()` or `PSA_CIPHER_FINISH_OUTPUT_SIZE()`.

`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 last part of 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. The first `(*output_length)` bytes of `output` contain the final output.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error:

- The operation state is not valid: it must be active, with an IV set if required for the algorithm.
- The library requires initializing by a call to `psa_crypto_init()`.

`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 a sufficient buffer 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_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_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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.

10.5.4 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: a 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_ALG_CCM_STAR_ANY_TAG` (macro)

A wildcard algorithm that permits the use of the key with CCM* as both an AEAD and an unauthenticated cipher algorithm.

Added in version 1.2.

```
#define PSA_ALG_CCM_STAR_ANY_TAG ((psa_algorithm_t)0x04c09300)
```

If a block-cipher key specifies `PSA_ALG_CCM_STAR_ANY_TAG` as its permitted algorithm, then the key can be used with the `PSA_ALG_CCM_STAR_NO_TAG` unauthenticated cipher, the `PSA_ALG_CCM` AEAD algorithm, and truncated `PSA_ALG_CCM` AEAD algorithms.

`PSA_CIPHER_ENCRYPT_OUTPUT_SIZE` (macro)

A sufficient output buffer size for `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: a value of type `psa_algorithm_t` 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)

A sufficient output buffer size for `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: a value of type `psa_algorithm_t` 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: a value of type `psa_algorithm_t` 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)

A sufficient buffer size for storing the IV generated by psa_cipher_generate_iv(), for any of the supported key types and cipher algorithms.

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

If the size of the IV buffer is at least this large, it is guaranteed that psa_cipher_generate_iv() will not fail due to an insufficient buffer size.

See also PSA_CIPHER_IV_LENGTH().

PSA_CIPHER_UPDATE_OUTPUT_SIZE (macro)

A sufficient output buffer size for psa_cipher_update(), in bytes.

```
#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: a value of type psa_algorithm_t 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 output 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: a value of type `psa_algorithm_t` 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 output buffer is at least this large, it is guaranteed that `psa_cipher_finish()` 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_FINISH_OUTPUT_MAX_SIZE`.

`PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE` (macro)

A sufficient output 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 */
```

If the size of the output buffer is at least this large, it is guaranteed that `psa_cipher_finish()` will not fail due to an insufficient buffer size.

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: a 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.

See also `PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE`.

`PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE` (macro)

The maximum block 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()`.

10.6 Authenticated encryption with associated data (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. A multi-part AEAD operation is used as follows:

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()`.

> **Note:**
>
> Using a multi-part interface to authenticated encryption raises specific issues.
>
> - Multi-part authenticated decryption produces intermediate results that are not authenticated. Revealing unauthenticated results, either directly or indirectly through the application's behavior, can compromise the confidentiality of all inputs that are encrypted with the same key. See the detailed warning.
> - 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.

10.6.1 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 affects the maximum length of the plaintext than can be encrypted or decrypted. If the nonce has length N, then the plaintext length $pLen$ is encoded in $L = 15 - N$ octets, this requires that $pLen < 2^{8L}$.

The value for L that is used with `PSA_ALG_CCM` depends on the function used to provide the nonce:

- A call to `psa_aead_encrypt()`, `psa_aead_decrypt()`, or `psa_aead_set_nonce()` will set $L = 15 - \texttt{nonce\_length}$. If the plaintext length cannot be encoded in L octets, then a `PSA_ERROR_INVALID_ARGUMENT` error is returned.

- A call to psa_aead_generate_nonce() on a multi-part cipher operation will select the smallest integer $L \geq 2$, where $pLen < 2^{8L}$, with $pLen$ being the plaintext_length provided to psa_aead_set_lengths(). The call to psa_aead_generate_nonce() will generate and return a random nonce of length $15 - L$ 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].

Usage in Zigbee

The CCM* algorithm is required by *zigbee Specification* [ZIGBEE].

- PSA_ALG_CCM, and its truncated variants, can be used to implement CCM* for non-zero tag lengths.
- For unauthenticated CCM*, with a zero-length tag, use the PSA_ALG_CCM_STAR_NO_TAG cipher algorithm.

See also Usage in Zigbee under PSA_ALG_CCM_STAR_NO_TAG.

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

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].

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

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* [RFC8439], 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 [RFC8439] variant.

Implementations must support 16-byte tags. It is recommended that truncated tag sizes are rejected.

Compatible key types

PSA_KEY_TYPE_CHACHA20

PSA_ALG_XCHACHA20_POLY1305 (macro)

The XChaCha20-Poly1305 AEAD algorithm.

Added in version 1.2.

```
#define PSA_ALG_XCHACHA20_POLY1305 ((psa_algorithm_t)0x05100600)
```

XChaCha20-Poly1305 is a variation of the ChaCha20-Poly1305 AEAD algorithm, but uses a 192-bit nonce. The larger nonce provides much lower probability of nonce misuse.

XChaCha20-Poly1305 requires a 24-byte nonce.

Implementations must support 16-byte tags. It is recommended that truncated tag sizes are rejected.

XChaCha20-Poly1305 is defined in *XChaCha: eXtended-nonce ChaCha and AEAD_XChaCha20_Poly1305* [XCHACHA].

Compatible key types

PSA_KEY_TYPE_XCHACHA20

PSA_ALG_ASCON_AEAD128 (macro)

The Ascon-AEAD128 AEAD algorithm.

Added in version 1.4.

```
#define PSA_ALG_ASCON_AEAD128 ((psa_algorithm_t)0x05100700)
```

There are two variants of Ascon-AEAD128 defined in *NIST Special Publication 800-232: Ascon-Based Lightweight Cryptography Standards for Constrained Devices* [SP800-232]:

- An implementation that supports Ascon-AEAD128 must provide the standard variant, using a 128-bit key. This is defined in [SP800-232] §4.1.
- An implementation can optionally also provide the nonce-masking variant, using a 256-bit key. This is defined in [SP800-232] §4.2.2.

The variant is selected based on the size of the key.

Both variants require a 128-bit (16 byte) nonce, which must not be reused with the same key.

Implementations must support 16-byte tags. Truncated tags of at least 4 bytes are permitted, but it is recommended that truncated tag sizes are at least 8 bytes. See [SP800-232] §4.2.1 and §4.3.R4.

Compatible key types

`PSA_KEY_TYPE_ASCON`

`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: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_AEAD(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 `aead_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()`.

Compatible key types

The resulting AEAD algorithm is compatible with the same key types as the AEAD algorithm used to construct it.

`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: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_AEAD(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()`.

Compatible key types

The resulting AEAD algorithm is compatible with the same key types as the AEAD algorithm used to construct it.

`PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG` (macro)

Macro to build an AEAD minimum-tag-length wildcard algorithm.

Added in version 1.1.

```
#define PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG(aead_alg, min_tag_length) \
    /* specification-defined value */
```

Parameters

`aead_alg`	An AEAD algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_AEAD(aead_alg)` is true.
`min_tag_length`	Desired minimum length of the authentication tag in bytes. This must be at least `1` and at most the largest permitted tag length of the algorithm.

Returns

The corresponding AEAD wildcard algorithm with the specified minimum tag length.

Unspecified if `aead_alg` is not a supported AEAD algorithm or if `min_tag_length` is less than `1` or too large for the specified AEAD algorithm.

Description

A key with a minimum-tag-length AEAD wildcard algorithm as permitted-algorithm policy can be used with all AEAD algorithms sharing the same base algorithm, and where the tag length of the specific algorithm is equal to or larger then the minimum tag length specified by the wildcard algorithm.

Note:

When setting the minimum required tag length to less than the smallest tag length permitted by the base algorithm, this effectively becomes an 'any-tag-length-permitted' policy for that base algorithm.

The AEAD algorithm with a default length tag can be recovered using `PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG()`.

Compatible key types

The resulting wildcard AEAD algorithm is compatible with the same key types as the AEAD algorithm used to construct it.

10.6.2 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 permit the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The AEAD algorithm to compute: a value of type `psa_algorithm_t` 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. The first `(*ciphertext_length)` bytes of `ciphertext` contain the output.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 `ciphertext` buffer is too small. `PSA_AEAD_ENCRYPT_OUTPUT_SIZE()` or `PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE()` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an AEAD algorithm. • `key` is not compatible with `alg`. • `nonce_length` is not valid for use with `alg` and `key`. • `additional_data_length` or `plaintext_length` are too large for `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not an AEAD algorithm. • `key` is not supported for use with `alg`. • `nonce_length` is not supported for use with `alg` and `key`. • `additional_data_length` or `plaintext_length` are too large for the implementation.
`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_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 permit the usage `PSA_KEY_USAGE_DECRYPT`.
`alg`	The AEAD algorithm to compute: a value of type `psa_algorithm_t` 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. The first `(*plaintext_length)` bytes of `plaintext` contain the output.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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_SIGNATURE`	The ciphertext is not authentic.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `plaintext` buffer is too small. `PSA_AEAD_DECRYPT_OUTPUT_SIZE()` or `PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE()` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an AEAD algorithm. • `key` is not compatible with `alg`. • `nonce_length` is not valid for use with `alg` and `key`. • `additional_data_length` or `ciphertext_length` are too large for `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not an AEAD algorithm. • `key` is not supported for use with `alg`. • `nonce_length` is not supported for use with `alg` and `key`. • `additional_data_length` or `plaintext_length` are too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

10.6.3 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` (typedef)

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 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 permit the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The AEAD algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_AEAD(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	The following conditions can result in this error: • `alg` is not an AEAD algorithm. • `key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not an AEAD algorithm. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

The sequence of operations to encrypt a message with authentication is as follows:

1. Allocate an AEAD 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()`.

After a successful call to `psa_aead_encrypt_setup()`, the operation is active, and 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()`.

If `psa_aead_encrypt_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_aead_abort()`.

See *Multi-part operations* on page 27.

`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 permit the usage `PSA_KEY_USAGE_DECRYPT`.
`alg`	The AEAD algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_AEAD(alg)` is true.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	The following conditions can result in this error: • `alg` is not an AEAD algorithm. • `key` is not compatible with `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not an AEAD algorithm. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

The sequence of operations to decrypt a message with authentication is as follows:

1. Allocate an AEAD 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()`.

After a successful call to `psa_aead_decrypt_setup()`, the operation is active, and 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()`.

If `psa_aead_decrypt_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_aead_abort()`.

See *Multi-part operations* on page 27.

`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 following conditions can result in this error: • 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. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`ad_length` or `plaintext_length` are too large for the chosen algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`ad_length` or `plaintext_length` are too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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 appropriate for the selected algorithm and key: • A sufficient output size is `PSA_AEAD_NONCE_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. • `PSA_AEAD_NONCE_MAX_SIZE` evaluates to a sufficient output size for any supported AEAD algorithm.
`nonce_length`	On success, the number of bytes of the generated nonce.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `(*nonce_length)` bytes of `nonce` contain the generated nonce.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be an active AEAD encryption operation, with no nonce set. • The operation state is not valid: this is an algorithm which requires `psa_aead_set_lengths()` to be called before setting the nonce. • The library requires initializing by a call to `psa_crypto_init()`.
`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 a sufficient buffer size.

PSA_ERROR_INSUFFICIENT_ENTROPY

PSA_ERROR_INSUFFICIENT_MEMORY

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

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.

Most algorithms generate a default-length nonce, as returned by `PSA_AEAD_NONCE_LENGTH()`. Some algorithms can return a shorter nonce from `psa_aead_generate_nonce()`, see the individual algorithm descriptions for details.

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 following conditions can result in this error: • The operation state is not valid: it must be active, with no nonce set. • The operation state is not valid: this is an algorithm which requires `psa_aead_set_lengths()` to be called before setting the nonce.

- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT`	`nonce_length` is not valid for the chosen algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`nonce_length` is not supported for use with the operation's algorithm and key.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 additional data until `psa_aead_verify()` succeeds.

	See the detailed warning.
PSA_ERROR_BAD_STATE	The following conditions can result in this error: • 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. • The library requires initializing by a call to psa_crypto_init().
PSA_ERROR_INVALID_ARGUMENT	Excess additional data: the total input length to psa_aead_update_ad() is greater than the additional data length that was previously specified with psa_aead_set_lengths(), or is too large for the chosen AEAD algorithm.
PSA_ERROR_NOT_SUPPORTED	The total additional data length is too large for the implementation.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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. The first `(*output_length)` of `output` contains the output data.

> **Warning**
>
> When decrypting, do not use the output until `psa_aead_verify()` succeeds.
>
> See the detailed warning.

`PSA_ERROR_BAD_STATE` — The following conditions can result in this error:

- The operation state is not valid: it must be active, have a nonce set, and have lengths set if required by the algorithm.
- The library requires initializing by a call to `psa_crypto_init()`.

`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 a sufficient buffer size.

`PSA_ERROR_INVALID_ARGUMENT` — The following conditions can result in this error:

- Incomplete additional data: the total length of input to `psa_aead_update_ad()` is less than the additional data length that was previously specified with `psa_aead_set_lengths()`.
- Excess input data: the total length of input to `psa_aead_update()` is greater than the plaintext length that was previously specified with `psa_aead_set_lengths()`, or is too large for the specific AEAD algorithm.

`PSA_ERROR_NOT_SUPPORTED` — The total input length is too large for the implementation.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

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().

> **Note:**
>
> 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() provides sufficient input, or a subsequent call to psa_aead_finish() or psa_aead_verify() indicates the end of the input. The amount of data that can be delayed in this way is bounded by the associated output size macro: PSA_AEAD_UPDATE_OUTPUT_SIZE(), PSA_AEAD_FINISH_OUTPUT_SIZE(), or PSA_AEAD_VERIFY_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. The first `(*tag_length)` bytes of `tag` contain the authentication tag.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be an active encryption operation with a nonce set. • The library requires initializing by a call to `psa_crypto_init()`.
`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 following conditions can result in this error: • Incomplete additional data: the total length of input to `psa_aead_update_ad()` is less than the additional data length that was previously specified with `psa_aead_set_lengths()`. • Incomplete plaintext: the total length of input to `psa_aead_update()` 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_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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 expected authentication tag.
`tag_length`	Size of the `tag` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. For a decryption operation, it is now safe to use the additional data and the plaintext output.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be an active decryption operation with a nonce set.

	• The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_SIGNATURE`	The calculated authentication tag does not match the value in `tag`.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • Incomplete additional data: the total length of input to `psa_aead_update_ad()` is less than the additional data length that was previously specified with `psa_aead_set_lengths()`. • Incomplete ciphertext: the total length of input to `psa_aead_update()` 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_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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()`.

Implementation 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`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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.

10.6.4 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: a 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_AEAD_ENCRYPT_OUTPUT_SIZE` (macro)

A sufficient ciphertext buffer size for `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: a value of type `psa_algorithm_t` 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 ciphertext 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)

A sufficient plaintext buffer size for `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: a value of type `psa_algorithm_t` 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 plaintext 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: a value of type `psa_algorithm_t` 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

If the size of the nonce buffer is at least this large, it is guaranteed that `psa_aead_generate_nonce()` will not fail due to an insufficient buffer size.

For most AEAD algorithms, `PSA_AEAD_NONCE_LENGTH()` evaluates to the exact size of the nonce generated by `psa_aead_generate_nonce()`.

See also `PSA_AEAD_NONCE_MAX_SIZE`.

`PSA_AEAD_NONCE_MAX_SIZE` (macro)

A sufficient buffer size for storing the nonce generated by `psa_aead_generate_nonce()`, for any of the supported key types and AEAD algorithms.

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

If the size of the nonce buffer is at least this large, it is guaranteed that `psa_aead_generate_nonce()` will not fail due to an insufficient buffer size.

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: a value of type `psa_algorithm_t` 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: a value of type `psa_algorithm_t` 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 */
```

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.

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: a value of type `psa_algorithm_t` 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 is the size of the tag output from `psa_aead_finish()`.

If the size of the tag buffer is at least this large, it is guaranteed that `psa_aead_finish()` will not fail due to an insufficient tag buffer size.

See also `PSA_AEAD_TAG_MAX_SIZE`.

`PSA_AEAD_TAG_MAX_SIZE` (macro)

A sufficient buffer size for storing the tag output by `psa_aead_finish()`, for any of the supported key types and AEAD algorithms.

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

If the size of the tag buffer is at least this large, it is guaranteed that `psa_aead_finish()` will not fail due to an insufficient buffer size.

See also `PSA_AEAD_TAG_LENGTH()`.

`PSA_AEAD_VERIFY_OUTPUT_SIZE` (macro)

A sufficient plaintext buffer size for `psa_aead_verify()`, in bytes.

```
#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: a value of type `psa_algorithm_t` 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 */
```

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 buffer size.

See also `PSA_AEAD_VERIFY_OUTPUT_SIZE()`.

10.7 Key wrapping

Key wrapping is the process of encrypting a key, so that the resulting ciphertext can be stored, or transported, in a form that maintains the confidentiality of the key material. Key unwrapping reverses this process, extracting the key from the ciphertext. Some key-wrapping schemes also provide integrity protection, to ensure that modification of the ciphertext can be detected.

Some key-wrapping algorithms operate on arbitrary data, and provide authenticated encryption that is specifically designed for key values. For example, the AES Key-wrap algorithm AES-KW. For this type of algorithm, the Crypto API provides a simple pair of functions, `psa_unwrap_key()` and `psa_wrap_key()`, that unwrap or wrap key data in the default export format. When using one of these key-wrapping algorithms, the key attributes are managed by the application.

> **Note:**
>
> Other key-wrapping schemes define both the format of the wrapped key material and the algorithm that is used to perform the wrapping. For example PKCS#8 defines *EncryptedPrivateKeyInfo*, which is also described in *Asymmetric Key Packages* [RFC5958]. Wrapped-key formats typically encode the key type and wrapping algorithm within the output data, and can also include other key attributes. This version of the Crypto API does not support these key-wrapping schemes, but this is planned for a future version.

10.7.1 Key-wrapping algorithms

`PSA_ALG_KW` (macro)

A key-wrapping algorithm based on the NIST Key Wrap (KW) mode of a block cipher.

Added in version 1.4.

```
#define PSA_ALG_KW ((psa_algorithm_t)0x0B400100)
```

KW is defined for block ciphers that have a 128-bit block size. The underlying block cipher is determined by the key type.

Keys to be wrapped must have a length equal to a multiple of the 'semi-block' size for the block cipher. That is, a multiple of 8 bytes.

To wrap keys that are not a multiple of the semi-block size, PSA_ALG_KWP can be used.

This is the NIST Key Wrap algorithm, using any block-cipher that operates on 128-bit blocks, as defined in *NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping* [SP800-38F]. A definition of AES-KW is also found in *Advanced Encryption Standard (AES) Key Wrap Algorithm* [RFC3394].

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

PSA_ALG_KWP (macro)

A key-wrapping algorithm based on the NIST Key Wrap with Padding (KWP) mode of a block cipher.

Added in version 1.4.

```
#define PSA_ALG_KWP ((psa_algorithm_t)0x0BC00200)
```

KWP is defined for block ciphers that have a 128-bit block size. The underlying block cipher is determined by the key type.

This algorithm can wrap a key of any length.

This is the NIST Key Wrap with Padding algorithm, using any block-cipher that operates on 128-bit blocks, as defined in *NIST Special Publication 800-38F: Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping* [SP800-38F]. A definition of AES-KWP is also found in *Advanced Encryption Standard (AES) Key Wrap with Padding Algorithm* [RFC5649].

Compatible key types

PSA_KEY_TYPE_AES

PSA_KEY_TYPE_ARIA

PSA_KEY_TYPE_CAMELLIA

PSA_KEY_TYPE_SM4

10.7.2 Key wrapping functions

psa_unwrap_key (function)

Unwrap and import a key using a specified wrapping key.

Added in version 1.4.

```
psa_status_t psa_unwrap_key(const psa_key_attributes_t * attributes,
                            psa_key_id_t wrapping_key,
                            psa_algorithm_t alg,
                            const uint8_t * data,
                            size_t data_length,
                            psa_key_id_t * key);
```

Parameters

`attributes`

The attributes for the new key.

The following attributes are required for all keys:

- The key type determines how the decrypted `data` buffer is interpreted.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default volatile lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

The following attributes are optional:

- If the key size is nonzero, it must be equal to the key size determined from `data`.

> **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.

`wrapping_key`

Identifier of the key to use for the unwrapping operation. It must permit the usage `PSA_KEY_USAGE_UNWRAP`.

`alg`

The key-wrapping algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_WRAP(alg)` is true.

`data`

Buffer containing the wrapped key data. The content of this buffer is unwrapped using the algorithm `alg`, and then interpreted according to the type declared in `attributes`.

`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_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`wrapping_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • The wrapping key does not have the `PSA_KEY_USAGE_UNWRAP` flag, or it does not permit the requested algorithm. • The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
`PSA_ERROR_INVALID_SIGNATURE`	The wrapped key data could not be authenticated.
`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 following conditions can result in this error: • `alg` is not a key-wrapping algorithm. • `wrapping_key` is not compatible with `alg`. • The key type is invalid. • The key size is nonzero, and is incompatible with the wrapped key data in `data`. • The key lifetime is invalid. • The key identifier is not valid for the key lifetime. • The key usage flags include invalid values. • The key's permitted-usage algorithm is invalid. • The key attributes, as a whole, are invalid. • The key data is not correctly formatted for the key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported or is not a key-wrapping algorithm. • `wrapping_key` is not supported for use with `alg`. • The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

The key is unwrapped and extracted from the provided `data` buffer. Its location, policy, and type are taken from `attributes`.

The wrapped key data determines the key size. :code:`psa_get_key_bits(attributes)` must either match the determined key size or be `0`.

Implementations must reject an attempt to unwrap a key if the determined key size is `0`.

> **Note:**
>
> A call to `psa_unwrap_key()` first applies the decryption procedure associated with the key-wrapping algorithm `alg`, using the `wrapping_key` key, to the supplied `data` buffer. The resulting plaintext is retained within the cryptoprocessor, and used with the provided `attributes` to create a key, as if they were inputs to `psa_import_key()`.

> **Note:**
>
> The Crypto API does not support asymmetric private key objects outside of a key pair. When unwrapping a private key, the corresponding key-pair type is created. If the imported key data does not contain the public key, then the implementation will reconstruct the public key from the private key as needed.

> **Implementation note**
>
> It is recommended that the implementation supports unwrapping any key data that can be produced by a call to `psa_wrap_key()`, with the same key-wrapping algorithm and key, and matching key attributes.
>
> It is recommended that implementations reject wrapped key data if it might be erroneous, for example, if it is the wrong type or is truncated.

`psa_wrap_key` (function)

Wrap and export a key using a specified wrapping key.

Added in version 1.4.

```
psa_status_t psa_wrap_key(psa_key_id_t wrapping_key,
                          psa_algorithm_t alg,
                          psa_key_id_t key,
                          uint8_t * data,
                          size_t data_size,
                          size_t * data_length);
```

Parameters

`wrapping_key`	Identifier of the key to use for the wrapping operation. It must permit the usage `PSA_KEY_USAGE_WRAP`.
`alg`	The key-wrapping algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_WRAP(alg)` is true.
`key`	Identifier of the key to wrap. It must permit the usage `PSA_KEY_USAGE_EXPORT`.
`data`	Buffer where the wrapped 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_WRAP_KEY_OUTPUT_SIZE(wrap_key_type, alg, type, bits)`, where `wrap_key_type` is the type of the wrapping key, `alg` is the key-wrapping algorithm, `type` is the type of the key being wrapped, and `bits` is the bit-size of the key being wrapped. • `PSA_WRAP_KEY_PAIR_MAX_SIZE` evaluates to the maximum wrapped output size of any supported key pair, in any supported combination of key-wrapping algorithm and wrapping-key type. • This API defines no maximum size for wrapped 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 wrapped key data.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `(*data_length)` bytes of `data` contain the wrapped key.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	The following conditions can result in this error: • `wrapping_key` is not a valid key identifier. • `key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • The wrapping key does not have the `PSA_KEY_USAGE_WRAP` flag, or it does not permit the requested algorithm. • The key to be wrapped does not have the `PSA_KEY_USAGE_EXPORT` flag.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `data` buffer is too small. `PSA_WRAP_KEY_OUTPUT_SIZE()` or `PSA_WRAP_KEY_PAIR_MAX_SIZE` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not a key-wrapping algorithm. • `wrapping_key` is not compatible with `alg`. • `key` has a size that is not valid for `alg`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error:

- `alg` is not supported or is not a key-wrapping algorithm.
- `wrapping_key` is not supported for use with `alg`.
- The storage location of `key` does not support export of the key.
- The implementation does not support export of keys with the type of `key`.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

Wrap a key from the key store into a data buffer using a specified key-wrapping algorithm and key-wrapping key. On success, the output contains the wrapped key value. The policy of the key to be wrapped must have the usage flag `PSA_KEY_USAGE_EXPORT` set.

The output of this function can be passed to `psa_unwrap_key()`, specifying the same algorithm and wrapping key, with the same attributes as `key`, to create an equivalent key object.

> **Note:**
>
> A call to `psa_wrap_key()` first evaluates the key data for `key`, as if `psa_export_key()` is called, but retaining the key data within the cryptoprocessor. If this succeeds, the encryption procedure associated with the key-wrapping algorithm `alg`, using the `wrapping_key` key, is applied to the key data. The resulting ciphertext is then returned.

10.7.3 Support macros

`PSA_WRAP_KEY_OUTPUT_SIZE` (macro)

Sufficient output buffer size for `psa_wrap_key()`.

Added in version 1.4.

```
#define PSA_WRAP_KEY_OUTPUT_SIZE(wrap_key_type, alg, key_type, key_bits) \
    /* implementation-defined value */
```

Parameters

`wrap_key_type`	A supported key-wrapping key type.
`alg`	A supported key-wrapping algorithm.
`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_wrap_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

See also `PSA_WRAP_KEY_PAIR_MAX_SIZE`.

`PSA_WRAP_KEY_PAIR_MAX_SIZE` (macro)

Sufficient buffer size for wrapping any asymmetric key pair.

Added in version 1.4.

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

This value must be a sufficient buffer size when calling `psa_wrap_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_WRAP_KEY_OUTPUT_SIZE()`.

10.8 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:

- Multiple key and non-key outputs to be produced from a single derivation operation object.
- Key and non-key outputs can be extracted from the key-derivation object, or compared with existing key and non-key values.
- Algorithms that require high-entropy secret inputs. For example `PSA_ALG_HKDF`.
- Algorithms that work with low-entropy secret inputs, or passwords. For example `PSA_ALG_PBKDF2_HMAC()`.

An implementation with *isolation* has the following properties:

- The intermediate state of the key derivation is not visible to the caller.
- If an output of the derivation is a non-exportable key, then this key cannot be recovered outside the isolation boundary.

- If an output of the derivation is compared using `psa_key_derivation_verify_bytes()` or `psa_key_derivation_verify_key()`, then the output is not visible to the caller.

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_key()` or `psa_key_derivation_key_agreement()` to provide the secret inputs, and `psa_key_derivation_input_bytes()` or `psa_key_derivation_input_integer()` to provide the non-secret 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 an output or verification function:
   - `psa_key_derivation_output_key()` or `psa_key_derivation_output_key_custom()` to create a derived key.
   - `psa_key_derivation_output_bytes()` to export the derived data.
   - `psa_key_derivation_verify_key()` to compare a derived key with an existing key value.
   - `psa_key_derivation_verify_bytes()` to compare derived data with a buffer.

   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.

10.8.1 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: a value of type `psa_algorithm_t` 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 (input keying material) 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.

> **Warning**
>
> HKDF processes the salt as follows: first hash it with `hash_alg` if the salt is longer than the block size of the hash algorithm; then pad with null bytes up to the block size. As a result, it is possible for distinct salt inputs to result in the same outputs. To ensure unique outputs, it is recommended to use a fixed length for salt values.

Each input may only be passed once.

Compatible key types

`PSA_KEY_TYPE_DERIVE` (for the secret key)
`PSA_KEY_TYPE_RAW_DATA` (for the other inputs)

`PSA_ALG_HKDF_EXTRACT` (macro)

Macro to build an HKDF-Extract algorithm.

Added in version 1.1.

```
#define PSA_ALG_HKDF_EXTRACT(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

The corresponding HKDF-Extract algorithm. For example, `PSA_ALG_HKDF_EXTRACT(PSA_ALG_SHA_256)` is HKDF-Extract using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is the Extract step of HKDF as specified by *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [RFC5869] §2.2.

This key-derivation algorithm uses the following inputs:

- `PSA_KEY_DERIVATION_INPUT_SALT` is the salt.
- `PSA_KEY_DERIVATION_INPUT_SECRET` is the input keying material used in the "extract" step.

The inputs are mandatory and must be passed in the order above. Each input may only be passed once.

> **Warning**
>
> HKDF-Extract is not meant to be used on its own. `PSA_ALG_HKDF` should be used instead if possible. `PSA_ALG_HKDF_EXTRACT` is provided as a separate algorithm for the sake of protocols that use it as a building block. It may also be a slight performance optimization in applications that use HKDF with the same salt and key but many different info strings.

> **Warning**
>
> HKDF processes the salt as follows: first hash it with `hash_alg` if the salt is longer than the block size of the hash algorithm; then pad with null bytes up to the block size. As a result, it is possible for distinct salt inputs to result in the same outputs. To ensure unique outputs, it is recommended to use a fixed length for salt values.

Compatible key types

`PSA_KEY_TYPE_DERIVE` (for the input keying material)
`PSA_KEY_TYPE_RAW_DATA` (for the salt)

`PSA_ALG_HKDF_EXPAND` (macro)

Macro to build an HKDF-Expand algorithm.

Added in version 1.1.

```
#define PSA_ALG_HKDF_EXPAND(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

The corresponding HKDF-Expand algorithm. For example, `PSA_ALG_HKDF_EXPAND(PSA_ALG_SHA_256)` is HKDF-Expand using HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is the Expand step of HKDF as specified by *HMAC-based Extract-and-Expand Key Derivation Function (HKDF)* [RFC5869] §2.3.

This key-derivation algorithm uses the following inputs:

- `PSA_KEY_DERIVATION_INPUT_SECRET` is the pseudorandom key (PRK).
- `PSA_KEY_DERIVATION_INPUT_INFO` is the info string.

The inputs are mandatory and must be passed in the order above. Each input may only be passed once.

> **Warning**
>
> HKDF-Expand is not meant to be used on its own. `PSA_ALG_HKDF` should be used instead if possible. `PSA_ALG_HKDF_EXPAND` is provided as a separate algorithm for the sake of protocols that use it as a building block. It may also be a slight performance optimization in applications that use HKDF with the same salt and key but many different info strings.

Compatible key types

`PSA_KEY_TYPE_DERIVE` (for the pseudorandom key)
`PSA_KEY_TYPE_RAW_DATA` (for the info string)

`PSA_ALG_SP800_108_COUNTER_HMAC` (macro)

Macro to build a NIST SP 800-108 conformant, counter-mode KDF algorithm based on HMAC.

Added in version 1.2.

```
#define PSA_ALG_SP800_108_COUNTER_HMAC(hash_alg) \
    /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

The corresponding key-derivation algorithm. For example, the counter-mode KDF using HMAC-SHA-256 is `PSA_ALG_SP800_108_COUNTER_HMAC(PSA_ALG_SHA_256)`.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is an HMAC-based, counter mode key-derivation function, using the construction recommended by *NIST Special Publication 800-108r1: Recommendation for Key Derivation Using Pseudorandom Functions* [SP800-108], §4.1.

This key-derivation algorithm uses the following inputs:

- `PSA_KEY_DERIVATION_INPUT_SECRET` is the secret input keying material, K_{IN}.
- `PSA_KEY_DERIVATION_INPUT_LABEL` is the $Label$. It is optional; if omitted, $Label$ is a zero-length string. If provided, it must not contain any null bytes.
- `PSA_KEY_DERIVATION_INPUT_CONTEXT` is the $Context$. It is optional; if omitted, $Context$ is a zero-length string.

Each input can only be passed once. Inputs must be passed in the order above.

This algorithm uses the output length as part of the derivation process. In the derivation this value is L, the required output size in bits. After setup, the initial capacity of the key-derivation operation is $2^{29} - 1$ bytes (`0x1fffffff`). The capacity can be set to a lower value by calling `psa_key_derivation_set_capacity()`.

When the first output is requested, the value of L is calculated as $L = 8 * cap$, where cap is the value of `psa_key_derivation_get_capacity()`. Subsequent calls to `psa_key_derivation_set_capacity()` are not permitted for this algorithm.

The derivation is constructed as described in [SP800-108] §4.1, with the iteration counter i and output length L encoded as big-endian, 32-bit values. The resulting output stream $K_1 \,||\, K_2 \,||\, K_3 \,||\, ...$ is computed as:

$$K_i = \text{HMAC}(K_{IN}, [i]_4 \,||\, Label \,||\, \texttt{0x00} \,||\, Context \,||\, [L]_4), \quad \text{for } i = 1, 2, 3, ...$$

Where $[x]_n$ is the big-endian, n-byte encoding of the integer x.

Compatible key types

`PSA_KEY_TYPE_HMAC` (for the secret key)
`PSA_KEY_TYPE_DERIVE` (for the secret key)
`PSA_KEY_TYPE_RAW_DATA` (for the other inputs)

`PSA_ALG_SP800_108_COUNTER_CMAC` (macro)

Macro to build a NIST SP 800-108 conformant, counter-mode KDF algorithm based on CMAC.

Added in version 1.2.

```
#define PSA_ALG_SP800_108_COUNTER_CMAC ((psa_algorithm_t)0x08000800)
```

This is a CMAC-based, counter mode key-derivation function, using the construction recommended by *NIST Special Publication 800-108r1: Recommendation for Key Derivation Using Pseudorandom Functions* [SP800-108], §4.1.

This key-derivation algorithm uses the following inputs:

- `PSA_KEY_DERIVATION_INPUT_SECRET` is the secret input keying material, K_{IN}. This must be a block-cipher key that is compatible with the CMAC algorithm, and must be input using `psa_key_derivation_input_key()`. See also `PSA_ALG_CMAC`.

- PSA_KEY_DERIVATION_INPUT_LABEL is the $Label$. It is optional; if omitted, $Label$ is a zero-length string. If provided, it must not contain any null bytes.
- PSA_KEY_DERIVATION_INPUT_CONTEXT is the $Context$. It is optional; if omitted, $Context$ is a zero-length string.

Each input can only be passed once. Inputs must be passed in the order above.

This algorithm uses the output length as part of the derivation process. In the derivation this value is L, the required output size in bits. After setup, the initial capacity of the key-derivation operation is $2^{29} - 1$ bytes (0x1fffffff). The capacity can be set to a lower value by calling psa_key_derivation_set_capacity().

When the first output is requested, the value of L is calculated as $L = 8 * cap$, where cap is the value of psa_key_derivation_get_capacity(). Subsequent calls to psa_key_derivation_set_capacity() are not permitted for this algorithm.

The derivation is constructed as described in [SP800-108] §4.1, with the following details:

- The iteration counter i and output length L are encoded as big-endian, 32-bit values.
- The mitigation to make the CMAC-based construction robust is implemented.

The resulting output stream $K_1 \,||\, K_2 \,||\, K_3 \,||\, ...$ is computed as:

$$K_0 = \text{CMAC}(K_{IN}, Label \,||\, \texttt{0x00} \,||\, Context \,||\, [L]_4\ )$$
$$K_i = \text{CMAC}(K_{IN}, [i]_4 \,||\, Label \,||\, \texttt{0x00} \,||\, Context \,||\, [L]_4 \,||\, K_0), \quad \text{for } i = 1, 2, 3, ...$$

Where $[x]_n$ is the big-endian, n-byte encoding of the integer x.

Compatible key types

PSA_KEY_TYPE_AES (for the secret key)
PSA_KEY_TYPE_ARIA (for the secret key)
PSA_KEY_TYPE_CAMELLIA (for the secret key)
PSA_KEY_TYPE_SM4 (for the secret key)
PSA_KEY_TYPE_RAW_DATA (for the other inputs)

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: a value of type psa_algorithm_t 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"`.

Compatible key types

`PSA_KEY_TYPE_DERIVE` (for the secret key)
`PSA_KEY_TYPE_RAW_DATA` (for the other inputs)

`PSA_ALG_TLS12_PSK_TO_MS` (macro)

Macro to build a TLS-1.2 PSK-to-MasterSecret algorithm.

Changed in version 1.1: Added step to support cipher-suites that include a key-exchange.

```
#define PSA_ALG_TLS12_PSK_TO_MS(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` 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_OTHER_SECRET is the other secret for the computation of the premaster secret. This input is optional; if omitted, it defaults to a string of null bytes with the same length as the secret (PSK) input.
- 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 other secret depends on the key exchange specified in the cipher suite:
  - For a plain PSK cipher suite ([RFC4279] §2), omit PSA_KEY_DERIVATION_INPUT_OTHER_SECRET.
  - For a DHE-PSK ([RFC4279] §3) or ECDHE-PSK cipher suite (*ECDHE_PSK Cipher Suites for Transport Layer Security (TLS)* [RFC5489] §2), the other secret should be the output of the PSA_ALG_FFDH or PSA_ALG_ECDH key agreement performed with the peer. The recommended way to pass this input is to use a key-derivation algorithm constructed as `PSA_ALG_KEY_AGREEMENT(ka_alg, PSA_ALG_TLS12_PSK_TO_MS(hash_alg))` and to call psa_key_derivation_key_agreement(). Alternatively, this input may be an output of psa_key_agreement() passed with psa_key_derivation_input_key(), or an equivalent input passed with psa_key_derivation_input_bytes() or psa_key_derivation_input_key().
  - For a RSA-PSK cipher suite ([RFC4279] §4), the other secret should be the 48-byte client challenge (the `PreMasterSecret` of [RFC5246] §7.4.7.1) concatenation of the TLS version and a 46-byte random string chosen by the client. On the server, this is typically an output of psa_asymmetric_decrypt() using PSA_ALG_RSA_PKCS1V15_CRYPT, passed to the key-derivation operation with psa_key_derivation_input_bytes().
- The label is `"master secret"` or `"extended master secret"`.

Compatible key types

PSA_KEY_TYPE_DERIVE (for the PSK)
PSA_KEY_TYPE_RAW_DATA (for the other inputs)

PSA_ALG_TLS12_ECJPAKE_TO_PMS (macro)

The TLS 1.2 ECJPAKE-to-PMS key-derivation algorithm.

Added in version 1.2.

```
#define PSA_ALG_TLS12_ECJPAKE_TO_PMS ((psa_algorithm_t)0x08000609)
```

This KDF is defined in *Elliptic Curve J-PAKE Cipher Suites for Transport Layer Security (TLS)* [TLS-ECJPAKE] §8.7. This specifies the use of a KDF to derive the TLS 1.2 session secrets from the output of EC J-PAKE over the secp256r1 Elliptic curve (the 256-bit curve in PSA_ECC_FAMILY_SECP_R1). EC J-PAKE operations can be performed using a PAKE operation, see *Password-authenticated key exchange (PAKE)* on page 338.

This KDF takes the shared secret K (an uncompressed EC point in case of EC J-PAKE) and calculates SHA256($K.x$).

This function takes a single input:

- PSA_KEY_DERIVATION_INPUT_SECRET is the shared secret K from EC J-PAKE. For secp256r1, the input is exactly 65 bytes.

  The shared secret can be obtained by calling psa_pake_get_shared_key() on a PAKE operation that is performing the EC J-PAKE algorithm. See *Password-authenticated key exchange (PAKE)* on page 338.

The 32-byte output has to be read in a single call to either psa_key_derivation_output_bytes(), psa_key_derivation_output_key(), or psa_key_derivation_output_key_custom(). The size of the output is defined as PSA_TLS12_ECJPAKE_TO_PMS_OUTPUT_SIZE.

Compatible key types

PSA_KEY_TYPE_DERIVE — the secret key is extracted from a PAKE operation by calling psa_pake_get_shared_key().

PSA_ALG_WPA3_SAE_H2E (macro)

The WPA3-SAE hash-to-element password token key-derivation algorithm.

Added in version 1.4.

```
#define PSA_ALG_WPA3_SAE_H2E(hash_alg) /* specification-defined value */
```

Parameters

hash_alg	A hash algorithm: a value of type psa_algorithm_t such that PSA_ALG_IS_HASH(hash_alg) is true. This includes PSA_ALG_ANY_HASH when specifying the algorithm in a key policy.

Description

This KDF is defined in *IEEE 802.11-2024: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications* [IEEE-802.11] §12.4.4. This specifies the hash-to-element procedures for deriving a WPA3-SAE password token from a network SSID and password. The resulting password token is then used during a WPA3-SAE PAKE operation.

This key-derivation algorithm uses the following inputs, which must be passed in the order given here:

- PSA_KEY_DERIVATION_INPUT_SALT is the network SSID.
- PSA_KEY_DERIVATION_INPUT_PASSWORD is the password.
- PSA_KEY_DERIVATION_INPUT_INFO is the password identifier. It is optional.

This key derivation algorithm can only be used to derive and output a single key, which is obtained by a call to psa_key_derivation_output_key(). The output has to be read as a key of type PSA_KEY_TYPE_WPA3_SAE_DH or PSA_KEY_TYPE_WPA3_SAE_ECC. Requesting any other key type, or calling psa_key_derivation_output_bytes(), returns an error status.

The hash_alg parameter to PSA_ALG_WPA3_SAE_H2E() determines the hash function used for the derivation. The key attributes of the output key indicate the elliptic curve or finite field group used for the derivation.

If the elliptic curve or finite field group specified in the key attributes is not compatible with the hash function used for the derivation, psa_key_derivation_output_bytes() returns PSA_ERROR_INVALID_ARGUMENT. See *WPA3-SAE cipher suites* on page 381.

WPA3-SAE password tokens on page 72 provides details of the derivation procedures.

> **Note:**
>
> To use a single password key with `PSA_ALG_WPA3_SAE_H2E` for any WPA3-SAE cipher suite, create the key with the wildcard `PSA_ALG_WPA3_SAE_ANY` permitted algorithm.

`PSA_ALG_PBKDF2_HMAC` (macro)

Macro to build a PBKDF2-HMAC password-hashing or key-stretching algorithm.

Added in version 1.1.

```
#define PSA_ALG_PBKDF2_HMAC(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

The corresponding PBKDF2-HMAC-XXX algorithm. For example, `PSA_ALG_PBKDF2_HMAC(PSA_ALG_SHA_256)` is the algorithm identifier for PBKDF2-HMAC-SHA-256.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

PBKDF2 is specified by *PKCS #5: Password-Based Cryptography Specification Version 2.1* [RFC8018] §5.2. This macro constructs a PBKDF2 algorithm that uses a pseudorandom function based on HMAC with the specified hash.

This key-derivation algorithm uses the following inputs, which must be provided in the following order:

- `PSA_KEY_DERIVATION_INPUT_COST` is the iteration count. This input step must be used exactly once.
- `PSA_KEY_DERIVATION_INPUT_SALT` is the salt. This input step must be used one or more times; if used several times, the inputs will be concatenated. This can be used to build the final salt from multiple sources, both public and secret (also known as pepper).
- `PSA_KEY_DERIVATION_INPUT_PASSWORD` is the password to be hashed. This input step must be used exactly once.

Compatible key types

`PSA_KEY_TYPE_DERIVE` (for password input)
`PSA_KEY_TYPE_PASSWORD` (for password input)
`PSA_KEY_TYPE_PEPPER` (for salt input)
`PSA_KEY_TYPE_RAW_DATA` (for salt input)
`PSA_KEY_TYPE_PASSWORD_HASH` (for key verification)

`PSA_ALG_PBKDF2_AES_CMAC_PRF_128` (macro)

The PBKDF2-AES-CMAC-PRF-128 password-hashing or key-stretching algorithm.

Added in version 1.1.

```
#define PSA_ALG_PBKDF2_AES_CMAC_PRF_128 ((psa_algorithm_t)0x08800200)
```

PBKDF2 is specified by *PKCS #5: Password-Based Cryptography Specification Version 2.1* [RFC8018] §5.2. This algorithm specifies the PBKDF2 algorithm using the AES-CMAC-PRF-128 pseudorandom function specified by [RFC4615]

This key-derivation algorithm uses the same inputs as `PSA_ALG_PBKDF2_HMAC()` with the same constraints.

Compatible key types

`PSA_KEY_TYPE_DERIVE` (for password input)
`PSA_KEY_TYPE_PASSWORD` (for password input)
`PSA_KEY_TYPE_PEPPER` (for salt input)
`PSA_KEY_TYPE_RAW_DATA` (for salt input)
`PSA_KEY_TYPE_PASSWORD_HASH` (for key verification)

10.8.2 Input step types

`psa_key_derivation_step_t` (typedef)

Encoding of the step of a key derivation.

```
typedef uint16_t psa_key_derivation_step_t;
```

> **Implementation note**
>
> It is recommended that the value `0` is not allocated as a valid key-derivation step.

`PSA_KEY_DERIVATION_INPUT_SECRET` (macro)

A high-entropy 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()`.

For some algorithms, a specific type of key is required. For example, see `PSA_ALG_SP800_108_COUNTER_CMAC`.

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 not permit a call to `psa_key_derivation_output_key()` or `psa_key_derivation_output_key_custom()`.

PSA_KEY_DERIVATION_INPUT_OTHER_SECRET (macro)

A high-entropy additional secret input for key derivation.

Added in version 1.1.

```
#define PSA_KEY_DERIVATION_INPUT_OTHER_SECRET \
    /* implementation-defined value */
```

This is typically the shared secret resulting from a key agreement obtained via `psa_key_derivation_key_agreement()`. It may alternatively be a key of type `PSA_KEY_TYPE_DERIVE` passed to `psa_key_derivation_input_key()`, or a direct input passed to `psa_key_derivation_input_bytes()`.

PSA_KEY_DERIVATION_INPUT_PASSWORD (macro)

A low-entropy secret input for password hashing or key stretching.

Added in version 1.1.

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

This is usually a key of type `PSA_KEY_TYPE_PASSWORD` passed to `psa_key_derivation_input_key()` or a direct input passed to `psa_key_derivation_input_bytes()` that is a password or passphrase. It can also be high-entropy secret, for example, a key of type `PSA_KEY_TYPE_DERIVE`, or the shared secret resulting from a key agreement.

If the secret is a direct input, the derivation operation cannot be used to derive keys: the operation will not permit a call to `psa_key_derivation_output_key()` or `psa_key_derivation_output_key_custom()`.

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` or `PSA_KEY_TYPE_PEPPER`.

`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`.

`PSA_KEY_DERIVATION_INPUT_COST` (macro)

A cost parameter for password hashing or key stretching.

Added in version 1.1.

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

This must be a direct input, passed to `psa_key_derivation_input_integer()`.

10.8.3 Key-derivation functions

`psa_key_derivation_operation_t` (typedef)

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 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 algorithm to compute. This must be one of the following: • A key-derivation algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_DERIVATION(alg)` is true. • A key-agreement and key-derivation algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_AGREEMENT(alg)` is true and `PSA_ALG_IS_RAW_KEY_AGREEMENT(alg)` is false.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`alg` is neither a key-derivation algorithm, nor a key-agreement and key-derivation algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`alg` is not supported or is not a key-derivation algorithm, or a key-agreement and key-derivation algorithm.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

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.

A key-agreement and key-derivation algorithm uses a key-agreement protocol to provide a shared secret which is used for the key derivation. See `psa_key_derivation_key_agreement()`.

The sequence of operations to derive a key is as follows:

1. Allocate a key-derivation 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_key_derivation_operation_t`, e.g. `PSA_KEY_DERIVATION_OPERATION_INIT`.
3. Call `psa_key_derivation_setup()` to specify the algorithm.
4. 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.
5. 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.
6. To derive a key, call `psa_key_derivation_output_key()` or `psa_key_derivation_output_key_custom()`. 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.
7. Clean up the key-derivation operation object with `psa_key_derivation_abort()`.

After a successful call to `psa_key_derivation_setup()`, the operation is active, and the application must eventually terminate the operation with a call to `psa_key_derivation_abort()`.

If `psa_key_derivation_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_key_derivation_abort()`.

See *Multi-part operations* on page 27.

`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`	Success. The maximum number of bytes that this key derivation can return is `(*capacity)`.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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()` or `psa_key_derivation_output_key_custom()` 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`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active. • The library requires initializing by a call to `psa_crypto_init()`.
`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_COMMUNICATION_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

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.

> **Note:**
>
> For some algorithms, the capacity value can affect the output of the key derivation. For example, see PSA_ALG_SP800_108_COUNTER_HMAC.

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_BAD_STATE	The following conditions can result in this error: • 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. • The library requires initializing by a call to psa_crypto_init().
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error: • step is not compatible with the operation's algorithm. • step does not permit direct inputs. • data_length is too small or too large for step in this particular algorithm.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error: • step is not supported with the operation's algorithm.

- `data_length` is is not supported for `step` in this particular algorithm.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

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_integer` (function)

Provide a numeric input for key derivation or key agreement.

Added in version 1.1.

```
psa_status_t psa_key_derivation_input_integer(psa_key_derivation_operation_t * operation,
                                              psa_key_derivation_step_t step,
                                              uint64_t value);
```

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.
`value`	The value of the numeric input.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • 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. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error:

- `step` is not compatible with the operation's algorithm.
- `step` does not permit numerical inputs.
- `value` is not valid for `step` in the operation's algorithm.

`PSA_ERROR_NOT_SUPPORTED` The following conditions can result in this error:

- `step` is not supported with the operation's algorithm.
- `value` is not supported for `step` in the operation's algorithm.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

Which inputs are required and in what order depends on the algorithm. However, when an algorithm requires a particular order, numeric inputs usually come first as they tend to be configuration parameters. Refer to the documentation of each key-derivation or key-agreement algorithm for information.

This function is used for inputs which are fixed-size non-negative integers.

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. The key must have an appropriate type for `step`, it must permit the usage `PSA_KEY_USAGE_DERIVE` or `PSA_KEY_USAGE_VERIFY_DERIVATION` (see note), and it must permit the algorithm used by the operation.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • 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. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • The key has neither the `PSA_KEY_USAGE_DERIVE` nor the `PSA_KEY_USAGE_VERIFY_DERIVATION` usage flag. • The key does not permit the operation's algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `step` is not compatible with the operation's algorithm. • `step` does not permit key inputs of the given type, or does not permit key inputs at all.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `step` is not supported with the operation's algorithm. • Key inputs of the given type are not supported for `step` in the operation's algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

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.

Note:

Once all inputs steps are completed, the following operations are permitted:

- `psa_key_derivation_output_bytes()` — if each input was either a direct input, or a key with usage flag `PSA_KEY_USAGE_DERIVE`.
- `psa_key_derivation_output_key()` or `psa_key_derivation_output_key_custom()` — if the input for

step PSA_KEY_DERIVATION_INPUT_SECRET or PSA_KEY_DERIVATION_INPUT_PASSWORD was a key with usage flag PSA_KEY_USAGE_DERIVE, and every other input was either a direct input or a key with usage flag PSA_KEY_USAGE_DERIVE.

- psa_key_derivation_verify_bytes()
- psa_key_derivation_verify_key()

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	Success. The first output_length bytes of output contain the derived data.
PSA_ERROR_BAD_STATE	The following conditions can result in this error: • The operation state is not valid: it must be active, with all required input steps complete. • The library requires initializing by a call to psa_crypto_init().
PSA_ERROR_NOT_PERMITTED	One of the inputs was a key whose policy did not permit PSA_KEY_USAGE_DERIVE.
PSA_ERROR_INSUFFICIENT_DATA	The operation's capacity was less than output_length bytes. In this case, the following occurs: • No output is written to the output buffer. • The operation's capacity is set to zero.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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.

A request to extract more data than the remaining capacity — `output_length > psa_key_derivation_get_capacity()` — fails with `PSA_ERROR_INSUFFICIENT_DATA`, and sets the remaining capacity to zero.

If the operation's capacity is zero, and `output_length` is zero, then it is IMPLEMENTATION DEFINED whether this function returns `PSA_SUCCESS` or `PSA_ERROR_INSUFFICIENT_DATA`.

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.

The following attributes are required for all keys:

- The key type. It must not be an asymmetric public key.
- The key size. It must be a valid size for the key type.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
  If the key type to be created is `PSA_KEY_TYPE_PASSWORD_HASH`, then the permitted-algorithm policy must be either the same as the current operation's algorithm, or `PSA_ALG_NONE`.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

> **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. For persistent keys, this is the key identifier defined in `attributes`. `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_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, with all required input steps complete. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • A `PSA_KEY_DERIVATION_INPUT_SECRET` or `PSA_KEY_DERIVATION_INPUT_PASSWORD` input step was neither provided through a key, nor the result of a key agreement. • One of the inputs was a key whose policy did not permit `PSA_KEY_USAGE_DERIVE`. • The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
`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. In this case, the following occurs: • No key is generated. • The operation's capacity is set to zero.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The key type is invalid, or is an asymmetric public-key type. • The key type is `PSA_KEY_TYPE_PASSWORD_HASH`, and the permitted-algorithm policy is not the same as the current operation's algorithm. • The key size is not valid for the key type. Implementations must reject an attempt to derive a key of size `0`. • The key lifetime is invalid. • The key identifier is not valid for the key lifetime. • The key usage flags include invalid values. • The key's permitted-usage algorithm is invalid. • The key attributes, as a whole, are invalid.
`PSA_ERROR_NOT_SUPPORTED`	The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_INSUFFICIENT_STORAGE`	

PSA_ERROR_COMMUNICATION_FAILURE

PSA_ERROR_CORRUPTION_DETECTED

PSA_ERROR_STORAGE_FAILURE

PSA_ERROR_DATA_CORRUPT

PSA_ERROR_DATA_INVALID

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.

A request that needs to extract more data than the remaining capacity fails with `PSA_ERROR_INSUFFICIENT_DATA`, and sets the remaining capacity to zero.

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. The key-derivation procedures for standard key-derivation algorithms are described in the *Key derivation* section of each key definition in *Key types* on page 53. Implementations can use other methods for implementation-specific algorithms.

For algorithms that take a `PSA_KEY_DERIVATION_INPUT_SECRET` or `PSA_KEY_DERIVATION_INPUT_PASSWORD` input step, 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.

> **Note:**
>
> This function is equivalent to calling `psa_key_derivation_output_key_custom()` with the production parameters `PSA_CUSTOM_KEY_PARAMETERS_INIT` and `custom_data_length == 0` (`custom_data` is ignored).

`psa_key_derivation_output_key_custom` (function)

Derive a key from an ongoing key-derivation operation with custom production parameters.

Added in version 1.3.

```
psa_status_t psa_key_derivation_output_key_custom(const psa_key_attributes_t * attributes,
                                                  psa_key_derivation_operation_t * operation,
                                                  const psa_custom_key_parameters_t * custom,
                                                  const uint8_t * custom_data,
                                                  size_t custom_data_length,
                                                  psa_key_id_t * key);
```

Parameters

`attributes`

The attributes for the new key.

The following attributes are required for all keys:

- The key type. It must not be an asymmetric public key.
- The key size. It must be a valid size for the key type.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
  If the key type to be created is `PSA_KEY_TYPE_PASSWORD_HASH`, then the permitted-algorithm policy must be either the same as the current operation's algorithm, or `PSA_ALG_NONE`.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

> **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.

`custom`

Customized production parameters for the key derivation.

When this is `PSA_CUSTOM_KEY_PARAMETERS_INIT` with `custom_data_length == 0`, this function is equivalent to `psa_key_derivation_output_key()`.

`custom_data`

A buffer containing additional variable-sized production parameters.

`custom_data_length`

Length of `custom_data` in bytes.

`key`

On success, an identifier for the newly created key. For persistent keys, this is the key identifier defined in `attributes`. `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_BAD_STATE`

The following conditions can result in this error:

- The operation state is not valid: it must be active, with all required input steps complete.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_NOT_PERMITTED`

The following conditions can result in this error:

	• A PSA_KEY_DERIVATION_INPUT_SECRET or PSA_KEY_DERIVATION_INPUT_PASSWORD input step was neither provided through a key, nor the result of a key agreement. • One of the inputs was a key whose policy did not permit PSA_KEY_USAGE_DERIVE. • The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
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. In this case, the following occurs: • No key is generated. • The operation's capacity is set to zero.
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error: • The key type is invalid, or is an asymmetric public-key type. • The key type is PSA_KEY_TYPE_PASSWORD_HASH, and the permitted-algorithm policy is not the same as the current operation's algorithm. • The key size is not valid for the key type. Implementations must reject an attempt to derive a key of size 0. • The key lifetime is invalid. • The key identifier is not valid for the key lifetime. • The key usage flags include invalid values. • The key's permitted-usage algorithm is invalid. • The key attributes, as a whole, are invalid. • The production parameters are invalid.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error: • The key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location. • The production parameters are not supported by the implementation.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_INSUFFICIENT_STORAGE	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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`.

This function operates in a similar way to `psa_key_derivation_output_key()`, but enables explicit production parameters to be provided when deriving a key. For example, the production parameters can be used to select an alternative key-derivation process, or configure additional key parameters. See `psa_key_derivation_output_key()` for the operation of this function with the default production parameters.

See `psa_custom_key_parameters_t` for a list of non-default production parameters. See the key type definitions in *Key types* on page 53 for details of the custom production parameters used for key derivation.

`psa_key_derivation_verify_bytes` (function)

Compare output data from a key-derivation operation to an expected value.

Added in version 1.1.

```
psa_status_t psa_key_derivation_verify_bytes(psa_key_derivation_operation_t * operation,
                                             const uint8_t * expected_output,
                                             size_t output_length);
```

Parameters

`operation`	The key-derivation operation object to read from.
`expected_output`	Buffer containing the expected derivation output.
`output_length`	Length of the expected output. This is also the number of bytes that will be read.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The output of the key-derivation operation matches `expected_output`.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, with all required input steps complete. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_SIGNATURE`	The output of the key-derivation operation does not match the value in `expected_output`.
`PSA_ERROR_INSUFFICIENT_DATA`	The operation's capacity was less than `output_length` bytes. In this case, the operation's capacity is set to zero.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

This function calculates output bytes from a key-derivation algorithm and compares those bytes to an expected value. If the key derivation's output is viewed as a stream of bytes, this function destructively reads `output_length` bytes from the stream before comparing them with `expected_output`. The operation's capacity decreases by the number of bytes read.

A request to extract more data than the remaining capacity — `output_length > psa_key_derivation_get_capacity()` — fails with `PSA_ERROR_INSUFFICIENT_DATA`, and sets the remaining capacity to zero.

If the operation's capacity is zero, and `output_length` is zero, then it is IMPLEMENTATION DEFINED whether this function returns `PSA_SUCCESS` or `PSA_ERROR_INSUFFICIENT_DATA`.

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()`.

Note:

A call to `psa_key_derivation_verify_bytes()` is functionally equivalent to the following code:

```
uint8_t tmp[output_length];
psa_key_derivation_output_bytes(operation, tmp, output_length);
if (memcmp(expected_output, tmp, output_length) != 0)
    return PSA_ERROR_INVALID_SIGNATURE;
```

However, calling `psa_key_derivation_verify_bytes()` works even if an input key's policy does not include `PSA_KEY_USAGE_DERIVE`.

Implementation note

Implementations must make the best effort to ensure that the comparison between the actual key-derivation output and the expected output is performed in constant time.

`psa_key_derivation_verify_key` (function)

Compare output data from a key-derivation operation to an expected value stored in a key.

Added in version 1.1.

```
psa_status_t psa_key_derivation_verify_key(psa_key_derivation_operation_t * operation,
                                           psa_key_id_t expected);
```

Parameters

`operation`	The key-derivation operation object to read from.
`expected`	A key of type `PSA_KEY_TYPE_PASSWORD_HASH` containing the expected output. The key must permit the usage `PSA_KEY_USAGE_VERIFY_DERIVATION`, and the permitted algorithm must match the operation's algorithm.

	The value of this key is typically computed by a previous call to `psa_key_derivation_output_key()` or `psa_key_derivation_output_key_custom()`.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The output of the key-derivation operation matches the `expected` key value.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, with all required input steps complete. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`expected` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The `expected` key does not have the `PSA_KEY_USAGE_VERIFY_DERIVATION` flag, or it does not permit the requested algorithm.
`PSA_ERROR_INVALID_SIGNATURE`	The output of the key-derivation operation does not match the value of the `expected` key.
`PSA_ERROR_INSUFFICIENT_DATA`	The operation's capacity was less than the length of the `expected` key. In this case, the operation's capacity is set to zero.
`PSA_ERROR_INVALID_ARGUMENT`	The key type is not `PSA_KEY_TYPE_PASSWORD_HASH`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

This function calculates output bytes from a key-derivation algorithm and compares those bytes to an expected value, provided as key of type `PSA_KEY_TYPE_PASSWORD_HASH`. If the key derivation's output is viewed as a stream of bytes, this function destructively reads the number of bytes corresponding to the length of the `expected` key from the stream before comparing them with the key value. The operation's capacity decreases by the number of bytes read.

A request that needs to extract more data than the remaining capacity fails with `PSA_ERROR_INSUFFICIENT_DATA`, and sets the remaining capacity to zero.

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()`.

> **Note:**
>
> A call to `psa_key_derivation_verify_key()` is functionally equivalent to exporting the `expected` key and calling `psa_key_derivation_verify_bytes()` on the result, except that it works when the key cannot be exported.

Implementation note

Implementations must make the best effort to ensure that the comparison between the actual key-derivation output and the expected output is performed in constant time.

`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`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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.

10.8.4 Support macros

`PSA_ALG_IS_KEY_DERIVATION_STRETCHING` (macro)

Whether the specified algorithm is a key-stretching or password-hashing algorithm.

Added in version 1.1.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

Description

A key-stretching or password-hashing algorithm is a key-derivation algorithm that is suitable for use with a low-entropy secret such as a password. Equivalently, it's a key-derivation algorithm that uses a `PSA_KEY_DERIVATION_INPUT_PASSWORD` input step.

`PSA_ALG_IS_HKDF` (macro)

Whether the specified algorithm is an HKDF algorithm (`PSA_ALG_HKDF(hash_alg)`).

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

Parameters

`alg`	An algorithm identifier: a 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_HKDF_EXTRACT` (macro)

Whether the specified algorithm is an HKDF-Extract algorithm (`PSA_ALG_HKDF_EXTRACT(hash_alg)`).

Added in version 1.1.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

`PSA_ALG_IS_HKDF_EXPAND` (macro)

Whether the specified algorithm is an HKDF-Expand algorithm (`PSA_ALG_HKDF_EXPAND(hash_alg)`).

Added in version 1.1.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

`PSA_ALG_IS_SP800_108_COUNTER_HMAC` (macro)

Whether the specified algorithm is a key-derivation algorithm constructed using `PSA_ALG_SP800_108_COUNTER_HMAC(hash_alg)`.

Added in version 1.2.

```
#define PSA_ALG_IS_SP800_108_COUNTER_HMAC(alg) \
    /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a key-derivation algorithm constructed using `PSA_ALG_SP800_108_COUNTER_HMAC()`, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported key-derivation algorithm identifier.

`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: a 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: a 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_ALG_IS_PBKDF2_HMAC` (macro)

Whether the specified algorithm is a PBKDF2-HMAC algorithm.

Added in version 1.1.

```
#define PSA_ALG_IS_PBKDF2_HMAC(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a PBKDF2-HMAC algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported key-derivation algorithm identifier.

`PSA_ALG_IS_WPA3_SAE_H2E` (macro)

Whether the specified algorithm is a WPA3-SAE hash-to-element key-derivation algorithm

Added in version 1.4.

```
#define PSA_ALG_IS_WPA3_SAE_H2E(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a WPA3-SAE hash-to-element 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`.

`PSA_TLS12_ECJPAKE_TO_PMS_OUTPUT_SIZE` (macro)

The size of the output from the TLS 1.2 ECJPAKE-to-PMS key-derivation algorithm, in bytes.

Added in version 1.2.

```
#define PSA_TLS12_ECJPAKE_TO_PMS_OUTPUT_SIZE 32
```

This value can be used when extracting the result of a key-derivation operation that was set up with the `PSA_ALG_TLS12_ECJPAKE_TO_PMS` algorithm.

10.9 Asymmetric signature

An asymmetric signature algorithm provides two functions:

- **Sign**: Calculate a message signature using a private, or secret, key.
- **Verify**: Check that a signature matches a message using a public key.

Successful verification indicates that the message signature was calculated using the private key that is associated with the public key.

In the Crypto API, an asymmetric-sign function requires an asymmetric key pair; and an asymmetric-verify function requires an asymmetric public key or key pair.

Signature schemes

The Crypto API supports the following signature schemes:

- *RSA signature algorithms* on page 280
- *ECDSA signature algorithms* on page 285
- *EdDSA signature algorithms* on page 289

Types of signature algorithm

There are three categories of asymmetric signature algorithm in the Crypto API:

- Hash-and-sign algorithms, that have two distinct phases:
  - Calculate a hash of the message
  - Calculate a signature over the hash

For these algorithms, the asymmetric signature API allows applications to either calculate the full message signature, or calculate the signature of a pre-computed hash. For example, this enables the application to use a multi-part hash operation to calculate the hash of a large message, prior to calculating or verifying a signature on the calculated hash.

The following algorithms are in this category:

`PSA_ALG_RSA_PKCS1V15_SIGN`

`PSA_ALG_RSA_PSS`

`PSA_ALG_RSA_PSS_ANY_SALT`

`PSA_ALG_ECDSA`

`PSA_ALG_DETERMINISTIC_ECDSA`

`PSA_ALG_ED25519PH`

`PSA_ALG_ED448PH`

- Message signature algorithms that do not separate the message processing from the signature calculations. This approach can provide better security against certain types of attack.

  For these algorithms, it is not possible to inject a pre-computed hash into the middle of the algorithm. An application can choose to calculate a message hash, and sign that instead of the message — but this is not functionally equivalent to signing the message, and eliminates the security benefits of signing the message directly.

  Some of these algorithms still permit the signature of a large message to be calculated, or verified, by providing the message data in fragments. This is possible when the algorithm only processes the message data once. See the individual algorithm descriptions for details.

  The following algorithms are in this category:

  `PSA_ALG_PURE_EDDSA`

  `PSA_ALG_EDDSA_CTX`

- Specialized signature algorithms, that use part of a standard signature algorithm within a specific protocol. It is recommended that these algorithms are only used for that purpose, with inputs as specified by the higher-level protocol. See the individual algorithm descriptions for details on their usage.

  The following algorithms are in this category:

  `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`

  `PSA_ALG_ECDSA_ANY`

Signature functions

The Crypto API provides several functions for calculating and verifying signatures:

- The single-part signature and verification functions, `psa_sign_message()` and `psa_verify_message()`, take a message as one of their inputs, and perform the sign or verify algorithm.

  These functions can be used on any hash-and-sign, or message signature, algorithms. See also `PSA_ALG_IS_SIGN_MESSAGE()`.

- The single-part functions, `psa_sign_hash()` and `psa_verify_hash()`, typically take a message hash as one of their inputs, and perform the sign or verify algorithm.

  These functions can be used on any hash-and-sign signature algorithm. It is recommended that the input to these functions is a hash, computed using the corresponding hash algorithm. To determine which hash algorithm to use, the macro `PSA_ALG_GET_HASH()` can be called on the signature algorithm identifier.

  These functions can also be used on the specialized signature algorithms, with a hash or encoded-hash as input. See also `PSA_ALG_IS_SIGN_HASH()`.

- Many modern signature algorithms have been designed to also accept a context parameter to provide domain separation. Version 1.4 of the Crypto API introduced four new functions that accept contexts: `psa_sign_message_with_context()`, `psa_sign_hash_with_context()`, `psa_verify_message_with_context()`, and `psa_verify_hash_with_context()`.

  If called with a zero-length context, these functions produce the same signature as the associated function without a context parameter.

  > **Note:**
  > If a signature scheme treats the absence of a context parameter differently to a zero-length context, the Crypto API defines distinct algorithm identifiers for the two variants. For example, when using a 255-bit key with EdDSA, `PSA_ALG_PURE_EDDSA` implements Ed25519 (without a context) and `PSA_ALG_EDDSA_CTX` implements Ed25519ctx (with a context, which can be zero-length). See *EdDSA signature algorithms* on page 289.

  It is an error to provide a non-zero-length context with an algorithm that does not accept contexts.

  Code written to be cryptographically agile can use the new functions, provided it guards against providing a non-zero-length context with an algorithm that does not support them.

  The `PSA_ALG_SIGN_SUPPORTS_CONTEXT()` macro can be used to determine if the implementation of an algorithm supports the use of non-zero-length contexts.

See *Asymmetric signature functions* on page 294.

10.9.1 RSA 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: a value of type `psa_algorithm_t` 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 hash-and-sign signature algorithm can be used with both the message and hash signature functions. RSA PKCS#1 v1.5 does not have a context parameter. However, the sign or verify with context functions can be used with a zero-length context.

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 the message digest, computed using the `hash_alg` hash algorithm.

Compatible key types

`PSA_KEY_TYPE_RSA_KEY_PAIR`

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (signature verification only)

`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 specialized signature algorithm can only be used with the `psa_sign_hash()` and `psa_verify_hash()` functions. RSA PKCS#1 v1.5 does not have a context parameter. However, `psa_sign_hash_with_context()` or `psa_verify_hash_with_context()` can be used with a zero-length context.

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 normally the DER encoding of the *DigestInfo* structure produced by step 2 in the message encoding algorithm, but it can be any byte string within the available length.

The wildcard key policy `PSA_ALG_RSA_PKCS1V15_SIGN(PSA_ALG_ANY_HASH)` also permits a key to be used with the `PSA_ALG_RSA_PKCS1V15_SIGN_RAW` signature algorithm.

Compatible key types

`PSA_KEY_TYPE_RSA_KEY_PAIR`

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (signature verification only)

`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: a value of type `psa_algorithm_t` 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 hash-and-sign signature algorithm can be used with both the message and hash signature functions. RSA PSS does not have a context parameter. However, the sign or verify with context functions can be used with a zero-length context.

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.
- When creating a signature, the salt length is equal to the length of the hash, or the largest possible salt length for the algorithm and key size if that is smaller than the hash length.
- When verifying a signature, the salt length must be equal to the length of the hash, or the largest possible salt length for the algorithm and key size if that is smaller than the hash length.
- The specified hash algorithm, `hash_alg`, is used to hash the input message, to create the salted hash, and for the mask generation.

When used with `psa_sign_hash()` or `psa_verify_hash()`, the provided `hash` parameter is the message digest, computed using the `hash_alg` hash algorithm.

> **Note:**
>
> The `PSA_ALG_RSA_PSS_ANY_SALT()` algorithm is equivalent to `PSA_ALG_RSA_PSS()` when creating a signature, but permits any salt length when verifying a signature.

Compatible key types

`PSA_KEY_TYPE_RSA_KEY_PAIR`

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (signature verification only)

`PSA_ALG_RSA_PSS_ANY_SALT` (macro)

The RSA PSS message signature scheme, with hashing. This variant permits any salt length for signature verification.

Added in version 1.1.

```
#define PSA_ALG_RSA_PSS_ANY_SALT(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` 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 hash-and-sign signature algorithm can be used with both the message and hash signature functions. RSA PSS does not have a context parameter. However, the sign or verify with context functions can be used with a zero-length context.

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.
- When creating a signature, the salt length is equal to the length of the hash, or the largest possible salt length for the algorithm and key size if that is smaller than the hash length.
- When verifying a signature, any salt length permitted by the RSASSA-PSS signature algorithm is accepted.
- The specified hash algorithm, `hash_alg`, is used to hash the input message, to create the salted hash, and for the mask generation.

When used with `psa_sign_hash()` or `psa_verify_hash()`, the provided `hash` parameter is the message digest, computed using the `hash_alg` hash algorithm.

> **Note:**
>
> The `PSA_ALG_RSA_PSS()` algorithm is equivalent to `PSA_ALG_RSA_PSS_ANY_SALT()` when creating a signature, but is strict about the permitted salt length when verifying a signature.

Compatible key types

`PSA_KEY_TYPE_RSA_KEY_PAIR`

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (signature verification only)

`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: a 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: a 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.

Description

This macro returns `1` for algorithms constructed using either `PSA_ALG_RSA_PSS()` or `PSA_ALG_RSA_PSS_ANY_SALT()`.

`PSA_ALG_IS_RSA_PSS_ANY_SALT` (macro)

Whether the specified algorithm is an RSA PSS signature algorithm that permits any salt length.

Added in version 1.1.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is an RSA PSS signature algorithm that permits any salt length, `0` otherwise.

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

Description

An RSA PSS signature algorithm that permits any salt length is constructed using `PSA_ALG_RSA_PSS_ANY_SALT()`.

See also `PSA_ALG_IS_RSA_PSS()` and `PSA_ALG_IS_RSA_PSS_STANDARD_SALT()`.

PSA_ALG_IS_RSA_PSS_STANDARD_SALT (macro)

Whether the specified algorithm is an RSA PSS signature algorithm that requires the standard salt length.

Added in version 1.1.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is an RSA PSS signature algorithm that requires the standard salt length, `0` otherwise.

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

Description

An RSA PSS signature algorithm that requires the standard salt length is constructed using `PSA_ALG_RSA_PSS()`.

See also `PSA_ALG_IS_RSA_PSS()` and `PSA_ALG_IS_RSA_PSS_ANY_SALT()`.

10.9.2 ECDSA signature algorithms

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: a value of type `psa_algorithm_t` 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 hash-and-sign signature algorithm can be used with both the message and hash signature functions. ECDSA does not have a context parameter. However, the sign or verify with context functions can be used with a zero-length context.

When used with `psa_sign_hash()` or `psa_verify_hash()`, the provided `hash` parameter is the message digest, computed using the `hash_alg` hash algorithm.

This algorithm is randomized: each invocation returns a different, equally valid signature.

The ECDSA 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 a big-endian m-octet string, where m is the integer for which $2^{8(m-1)} \leq q < 2^{8m}$, and q is the order of the elliptic curve.

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.

> **Note:**
>
> A verifier cannot 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.

When PSA_ALG_ECDSA(hash_alg) is used as a permitted algorithm in a key policy, this permits:

- PSA_ALG_ECDSA(hash_alg) as the algorithm in a call to any signing function.
- PSA_ALG_ECDSA(hash_alg) or PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) as the algorithm in a call to any signature verification function.

Compatible key types

PSA_KEY_TYPE_ECC_KEY_PAIR(family)

PSA_KEY_TYPE_ECC_PUBLIC_KEY(family) (signature verification only)

where family is a Weierstrass Elliptic curve family. That is, one of the following values:

- PSA_ECC_FAMILY_SECT_XX
- PSA_ECC_FAMILY_SECP_XX
- PSA_ECC_FAMILY_FRP
- PSA_ECC_FAMILY_BRAINPOOL_P_R1

PSA_ALG_ECDSA_ANY (macro)

The randomized ECDSA signature scheme, without hashing.

```
#define PSA_ALG_ECDSA_ANY ((psa_algorithm_t) 0x06000600)
```

This specialized signature algorithm can only be used with the psa_sign_hash() and psa_verify_hash() functions. ECDSA does not have a context parameter. However, psa_sign_hash_with_context() or psa_verify_hash_with_context() can be used with a zero-length context.

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.

> **Warning**
>
> This algorithm is only recommended to sign or verify a sequence of bytes that are a pre-computed hash. Note that the input is padded with zeros on the left or truncated on the right as required to fit the curve size.

This algorithm cannot be used with the wildcard key policy `PSA_ALG_ECDSA(PSA_ALG_ANY_HASH)`. It is only permitted when `PSA_ALG_ECDSA_ANY` is the key's permitted-algorithm policy.

Compatible key types

`PSA_KEY_TYPE_ECC_KEY_PAIR(family)`

`PSA_KEY_TYPE_ECC_PUBLIC_KEY(family)` (signature verification only)

where `family` is a Weierstrass Elliptic curve family. That is, one of the following values:

- `PSA_ECC_FAMILY_SECT_XX`
- `PSA_ECC_FAMILY_SECP_XX`
- `PSA_ECC_FAMILY_FRP`
- `PSA_ECC_FAMILY_BRAINPOOL_P_R1`

`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: a value of type `psa_algorithm_t` 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 hash-and-sign signature algorithm can be used with both the message and hash signature functions. ECDSA does not have a context parameter. However, the sign or verify with context functions can be used with a zero-length context.

When used with `psa_sign_hash()` or `psa_verify_hash()`, the provided `hash` parameter is the message digest, computed using the `hash_alg` hash algorithm.

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`.

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.

> **Note:**
>
> A verifier cannot 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.

When PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) is used as a permitted algorithm in a key policy, this permits:

- PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) as the algorithm in a call to any signing function.
- PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) or PSA_ALG_ECDSA(hash_alg) as the algorithm in a call to any signature verification function.

Compatible key types

PSA_KEY_TYPE_ECC_KEY_PAIR(family)
PSA_KEY_TYPE_ECC_PUBLIC_KEY(family) (signature verification only)

where family is a Weierstrass Elliptic curve family. That is, one of the following values:

- PSA_ECC_FAMILY_SECT_XX
- PSA_ECC_FAMILY_SECP_XX
- PSA_ECC_FAMILY_FRP
- PSA_ECC_FAMILY_BRAINPOOL_P_R1

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: a 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: a 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: a 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()`.

10.9.3 EdDSA signature algorithms

The PureEdDSA and HashEdDSA digital signature algorithms are defined by *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032]. They are used with the Edwards25519 and Edwards448 elliptic curve keys, see `PSA_ECC_FAMILY_TWISTED_EDWARDS`.

- PureEdDSA is a set of message-signing algorithms, that cannot be split into a hash step, followed by a signature or verification step.
- HashEdDSA is a pair of hash-and-sign algorithms, with a specified hash algorithm associated with each key size.

Both PureEdDSA and HashEdDSA can be used with contexts, which enables domain-separation when signatures are made of different message structures with the same key. For EdDSA, the context is an arbitrary byte string between zero and 255 bytes in length.

The development of EdDSA resulted in a total of five distinct algorithms:

- Ed25519: the original PureEdDSA algorithm for the Edwards25519 curve, which does not accept a context.
- Ed25519ctx: a second PureEdDSA algorithm for the Edwards25519 curve, with a context parameter.
- Ed448: the PureEdDSA algorithm for the Edwards448 curve, with a context parameter.
- Ed25519ph: the HashEdDSA algorithm for the Edwards25519 curve, with a context parameter.
- Ed448ph: the HashEdDSA algorithm for the Edwards448 curve, with a context parameter.

Table 16 shows the algorithm identifiers in the Crypto API, and how they are used to select the appropriate EdDSA algorithm.

Table 16 EdDSA algorithm identifiers

Algorithm identifier	With 255-bit key	With 448-bit key	Sign/verify hash	Support non-zero-length context
`PSA_ALG_PURE_EDDSA`	Ed25519	Ed448	No	No
`PSA_ALG_ED25519PH`	Ed25519ph	*Invalid*	Yes	Yes
`PSA_ALG_ED448PH`	*Invalid*	Ed448ph	Yes	Yes
`PSA_ALG_EDDSA_CTX`	Ed25519ctx	Ed448	No	Yes

Note:

Ed25519ctx produces a distinct signature to Ed25519, even with a zero-length context.

`PSA_ALG_PURE_EDDSA` (macro)

Edwards-curve digital signature algorithm without pre-hashing (PureEdDSA), with zero-length context.

Added in version 1.1.

```
#define PSA_ALG_PURE_EDDSA ((psa_algorithm_t) 0x06000800)
```

This message-signature algorithm can be used with the `psa_sign_message()` and `psa_verify_message()` functions. With a zero-length context, `PSA_ALG_PURE_EDDSA` can also be used with the `psa_sign_message_with_context()` and `psa_verify_message_with_context()` functions. It cannot be used to sign hashes.

This is the PureEdDSA digital signature algorithm defined by *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032], with zero-length context.

PureEdDSA requires an elliptic curve key on a twisted Edwards curve (see `PSA_ECC_FAMILY_TWISTED_EDWARDS`). The following curves are supported:

- Edwards25519: the Ed25519 algorithm is computed. The output signature is a 64-byte string: the concatenation of R and S as defined by [RFC8032] §5.1.6.
- Edwards448: the Ed448 algorithm is computed, with a zero-length context. The output signature is a 114-byte string: the concatenation of R and S as defined by [RFC8032] §5.2.6.

Note:

To sign or verify the pre-computed hash of a message using EdDSA, the HashEdDSA algorithms (PSA_ALG_ED25519PH and PSA_ALG_ED448PH) can be used. The signature produced by HashEdDSA is distinct from that produced by PureEdDSA.

Note:

To sign or verify a message with a non-zero-length context using PureEdDSA, use the PSA_ALG_EDDSA_CTX algorithm.

With an Edwards25519 curve key, PSA_ALG_EDDSA_CTX with a zero-length context creates different signatures to PSA_ALG_PURE_EDDSA.

Compatible key types

PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_TWISTED_EDWARDS)

PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_TWISTED_EDWARDS) (signature verification only)

PSA_ALG_EDDSA_CTX (macro)

Edwards-curve digital signature algorithm without pre-hashing (PureEdDSA), with a context.

Added in version 1.4.

```
#define PSA_ALG_EDDSA_CTX ((psa_algorithm_t) 0x06000A00)
```

This message-signature algorithm can be used with both the message and message with context signature functions. It cannot be used to sign hashes.

This is the PureEdDSA digital signature algorithm defined by *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032], with a context parameter. The context parameter can be between zero and 255 bytes in length.

PureEdDSA requires an elliptic curve key on a twisted Edwards curve (see PSA_ECC_FAMILY_TWISTED_EDWARDS). The following curves are supported:

- Edwards25519: the Ed25519ctx algorithm is computed. The output signature is a 64-byte string: the concatenation of R and S as defined by [RFC8032] §5.1.6.
- Edwards448: the Ed448 algorithm is computed, with a zero-length context. The output signature is a 114-byte string: the concatenation of R and S as defined by [RFC8032] §5.2.6.

To use a non-zero-length context, use the message-signature functions that accept a context parameter, psa_sign_message_with_context() and psa_verify_message_with_context() The psa_sign_message() and psa_verify_message() functions use a zero-length context when computing or verifying signatures.

Note:

To sign or verify the pre-computed hash of a message using EdDSA, the HashEdDSA algorithms (PSA_ALG_ED25519PH and PSA_ALG_ED448PH) can be used. The signature produced by HashEdDSA is

distinct from that produced by PureEdDSA.

Note:

With an Edwards25519 curve key, `PSA_ALG_EDDSA_CTX` with a zero-length context creates different signatures to `PSA_ALG_PURE_EDDSA`.

Usage

This is a message signing algorithm. To calculate a signature, use one of the following approaches:

- Call `psa_sign_message()` or `psa_sign_message_with_context()` with the message.

Verifying a signature is similar, using `psa_verify_message()` or `psa_verify_message_with_context()`.

Compatible key types

`PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_TWISTED_EDWARDS)`

`PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_TWISTED_EDWARDS)` (signature verification only)

`PSA_ALG_ED25519PH` (macro)

Edwards-curve digital signature algorithm with pre-hashing (HashEdDSA), using the Edwards25519 curve.

Added in version 1.1.

```
#define PSA_ALG_ED25519PH ((psa_algorithm_t) 0x0600090B)
```

This hash-and-sign signature algorithm can be used with both the message and hash signature functions.

This calculates the Ed25519ph algorithm as specified in *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032] §5.1, and requires an Edwards25519 curve key.

The pre-hash function is SHA-512, see `PSA_ALG_SHA_512`. When used to sign or verify a hash, the `hash` parameter is the SHA-512 message digest.

The signature functions without a context parameter use a zero-length context when computing or verifying signatures. To use a non-zero-length context, use the signature functions that accept a context parameter, such as `psa_sign_hash_with_context()` or `psa_verify_message_with_context()` The context parameter can be between zero and 255 bytes in length.

Implementation note

When used to sign or verify a hash, the `hash` parameter to the call should be used as $\mathrm{PH}(M)$ in the algorithms defined in [RFC8032] §5.1.

Usage

This is a hash-and-sign algorithm. To calculate a signature, use one of the following approaches:

- Call `psa_sign_message()` or `psa_sign_message_with_context()` with the message.

- Calculate the SHA-512 hash of the message with `psa_hash_compute()`, or with a multi-part hash operation, using the hash algorithm `PSA_ALG_SHA_512`. Then sign the calculated hash with `psa_sign_hash()` or `psa_sign_hash_with_context()`.

Verifying a signature is similar, using one of the following approaches:

- Call `psa_verify_message()`, or `psa_verify_message_with_context()` with the message.
- Calculate the SHA-512 hash of the message with `psa_hash_compute()`, or with a multi-part hash operation, using the hash algorithm `PSA_ALG_SHA_512`. Then sign the calculated hash with `psa_verify_hash()` or `psa_verify_hash_with_context()`.

Compatible key types

`PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_TWISTED_EDWARDS)`
`PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_TWISTED_EDWARDS)` (signature verification only)

`PSA_ALG_ED448PH` (macro)

Edwards-curve digital signature algorithm with pre-hashing (HashEdDSA), using the Edwards448 curve.

Added in version 1.1.

```
#define PSA_ALG_ED448PH ((psa_algorithm_t) 0x06000915)
```

This hash-and-sign signature algorithm can be used with both the message and hash signature functions.

This calculates the Ed448ph algorithm as specified in *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032] §5.2, and requires an Edwards448 curve key.

The pre-hash function is the first 64 bytes of the output from SHAKE256, see `PSA_ALG_SHAKE256_512`. When used to sign or verify a hash, the `hash` parameter is the truncated SHAKE256 message digest.

The signature functions without a context parameter use a zero-length context when computing or verifying signatures. To use a non-zero-length context, use the signature functions that accept a context parameter, for example, `psa_sign_hash_with_context()` or `psa_verify_message_with_context()` The context parameter can be between zero and 255 bytes in length.

> **Implementation note**
>
> When used to sign or verify a hash, the `hash` parameter to the call should be used as PH(M) in the algorithms defined in [RFC8032] §5.2.

Usage

This is a hash-and-sign algorithm. To calculate a signature, use one of the following approaches:

- Call `psa_sign_message()`, or `psa_sign_message_with_context()` with the message.
- Calculate the first 64 bytes of the SHAKE256 output of the message with `psa_hash_compute()`, or with a multi-part hash operation, using the hash algorithm `PSA_ALG_SHAKE256_512`. Then sign the calculated hash with `psa_sign_hash()` or `psa_sign_hash_with_context()`.

Verifying a signature is similar, using one of the following approaches:

- Call psa_verify_message(), or psa_verify_message_with_context() with the message.
- Calculate the first 64 bytes of the SHAKE256 output of the message with psa_hash_compute(), or with a multi-part hash operation, using the hash algorithm PSA_ALG_SHAKE256_512. Then sign the calculated hash with psa_verify_hash() or psa_verify_hash_with_context().

Compatible key types

PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_TWISTED_EDWARDS)

PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_TWISTED_EDWARDS) (signature verification only)

PSA_ALG_IS_HASH_EDDSA (macro)

Whether the specified algorithm is HashEdDSA.

Added in version 1.1.

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

Parameters

alg	An algorithm identifier: a value of type psa_algorithm_t.

Returns

1 if alg is a HashEdDSA algorithm, 0 otherwise.

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

10.9.4 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 permit the usage PSA_KEY_USAGE_SIGN_MESSAGE.
alg	An asymmetric signature algorithm: a value of type psa_algorithm_t such that PSA_ALG_IS_SIGN_MESSAGE(alg) is true.
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`	Success. The first `(*signature_length)` bytes of `signature` contain the signature value.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits signing a message. • `key` is not an asymmetric key pair, that is compatible with `alg`. • `input_length` is too large for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported, or is not an asymmetric signature algorithm that permits signing a message. • `key` is not supported for use with `alg`. • `input_length` is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

If the algorithm has a context parameter, a zero-length context is used. To provide a context value, use `psa_sign_message_with_context()` instead.

> **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_sign_message_with_context` (function)

Sign a message with a private key using a supplied context. For hash-and-sign algorithms, this includes the hashing step.

Added in version 1.4.

```
psa_status_t psa_sign_message_with_context(psa_key_id_t key,
                                           psa_algorithm_t alg,
                                           const uint8_t * input,
                                           size_t input_length,
                                           const uint8_t * context,
                                           size_t context_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 permit the usage `PSA_KEY_USAGE_SIGN_MESSAGE`.
`alg`	An asymmetric signature algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_MESSAGE(alg)` is true.
`input`	The input message to sign.
`input_length`	Size of the `input` buffer in bytes.
`context`	The context to use for this signature.
`context_length`	Size of the `context` 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`	Success. The first `(*signature_length)` bytes of `signature` contain the signature value.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits signing a message with a non-zero-length context. • `key` is not an asymmetric key pair, that is compatible with `alg`. • `input_length` is too large for the algorithm and key type. • `context_length` is not valid for the algorithm and key type. • `context` is not a valid input value for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported, or is not an asymmetric signature algorithm that permits signing a message. • `key` is not supported for use with `alg`. • The implementation does not support this value of `context_length` for `alg`. • `input_length` is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

If a context parameter is not required, `psa_sign_message()` can be used instead.

> **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_with_context()`. `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. For hash-and-sign algorithms, this includes the hashing step.

```
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 permit the usage `PSA_KEY_USAGE_VERIFY_MESSAGE`.
`alg`	An asymmetric signature algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_MESSAGE(alg)` is true.
`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`	Success. The signature is valid.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	`signature` is not the result of signing the `input` message with algorithm `alg` using the private key corresponding to `key`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits verifying a message. • `key` is not a public key or an asymmetric key pair, that is compatible with `alg`. • `input_length` is too large for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported, or is not an asymmetric signature algorithm that permits verifying a message. • `key` is not supported for use with `alg`. • `input_length` is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

If the algorithm has a context parameter, a zero-length context is used. To provide a context value, use `psa_verify_message_with_context()` instead.

> **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_verify_message_with_context` (function)

Verify the signature of a message with a public key and a supplied context. For hash-and-sign algorithms, this includes the hashing step.

Added in version 1.4.

```
psa_status_t psa_verify_message_with_context(psa_key_id_t key,
                                             psa_algorithm_t alg,
                                             const uint8_t * input,
                                             size_t input_length,
                                             const uint8_t * context,
                                             size_t context_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 permit the usage `PSA_KEY_USAGE_VERIFY_MESSAGE`.
`alg`	An asymmetric signature algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_MESSAGE(alg)` is true.
`input`	The message whose signature is to be verified.
`input_length`	Size of the `input` buffer in bytes.
`context`	The context to use for this signature.
`context_length`	Size of the `context` 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`	Success. The signature is valid.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	`signature` is not the result of signing the `input` message with algorithm `alg` using the private key corresponding to `key`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits verifying a message with a non-zero-length context. • `key` is not a public key or an asymmetric key pair, that is compatible with `alg`. • `input_length` is too large for the algorithm and key type. • `context_length` is not valid for the algorithm and key type. • `context` is not a valid input value for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported, or is not an asymmetric signature algorithm that permits verifying a message. • `key` is not supported for use with `alg`. • The implementation does not support this value of `context_length` for `alg`. • `input_length` is too large for the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

If a context parameter is not required, `psa_verify_message()` can be used instead.

> **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_with_context()`. `PSA_ALG_GET_HASH(alg)` can be used to determine the hash algorithm to use.

`psa_sign_hash` (function)

Sign a pre-computed 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 permit the usage `PSA_KEY_USAGE_SIGN_HASH`.
`alg`	An asymmetric signature algorithm that separates the hash and sign operations: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_HASH(alg)` is true.
`hash`	The input to sign. This is usually the hash of a message. See the description of this function, or the description of individual signature algorithms, for details of the 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`	Success. The first `(*signature_length)` bytes of `signature` contain the signature value.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits signing

	a pre-computed hash. • `key` is not an asymmetric key pair, that is compatible with `alg`. • `hash_length` is not valid for the algorithm and key type. • `hash` is not a valid input value for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported, or is not an asymmetric signature algorithm that permits signing a pre-computed hash. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

For hash-and-sign signature algorithms, the `hash` input to this function is the hash of the message to sign. The algorithm used to calculate this hash is encoded in the signature algorithm. For such algorithms, `hash_length` must equal the length of the hash output: `hash_length == PSA_HASH_LENGTH(PSA_ALG_GET_HASH(alg))`.

Specialized signature algorithms can apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. For example, see `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`.

If the algorithm has a context parameter, a zero-length context is used. To provide a context value, use `psa_sign_hash_with_context()` instead.

`psa_sign_hash_with_context` (function)

Sign a pre-computed hash with a private key and a supplied context.

Added in version 1.4.

```
psa_status_t psa_sign_hash_with_context(psa_key_id_t key,
                                        psa_algorithm_t alg,
                                        const uint8_t * hash,
                                        size_t hash_length,
                                        const uint8_t * context,
                                        size_t context_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 permit the usage `PSA_KEY_USAGE_SIGN_HASH`.
`alg`	An asymmetric signature algorithm that separates the hash and sign operations: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_HASH(alg)` is true.
`hash`	The input to sign. This is usually the hash of a message. See the description of this function, or the description of individual signature algorithms, for details of the acceptable inputs.
`hash_length`	Size of the `hash` buffer in bytes.
`context`	The context to use for this signature.
`context_length`	Size of the `context` 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`	Success. The first `(*signature_length)` bytes of `signature` contain the signature value.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits signing a pre-computed hash with a context. • `key` is not an asymmetric key pair, that is compatible with `alg`. • `hash_length` is not valid for the algorithm and key type. • `hash` is not a valid input value for the algorithm and key type. • `context_length` is not valid for the algorithm and key type. • `context` is not a valid input value for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error:

- `alg` is not supported, or is not an asymmetric signature algorithm that permits signing a pre-computed hash.
- The implementation does not support this value of `context_length` for `alg`.
- `key` is not supported for use with `alg`.

`PSA_ERROR_INSUFFICIENT_ENTROPY`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

For hash-and-sign signature algorithms, the `hash` input to this function is the hash of the message to sign. The algorithm used to calculate this hash is encoded in the signature algorithm. For such algorithms, `hash_length` must equal the length of the hash output: `hash_length == PSA_HASH_LENGTH(PSA_ALG_GET_HASH(alg))`.

Specialized signature algorithms can apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. For example, see `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`.

If a context parameter is not required, `psa_sign_hash()` can be used instead.

`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 permit the usage `PSA_KEY_USAGE_VERIFY_HASH`.

`alg` — An asymmetric signature algorithm that separates the hash and sign operations: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_HASH(alg)` is true.

`hash` — The input whose signature is to be verified. This is usually the hash of a message.

See the description of this function, or the description of individual signature algorithms, for details of the 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`	Success. The signature is valid.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	`signature` is not the result of signing `hash` with algorithm `alg` using the private key corresponding to `key`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits verifying a pre-computed hash. • `key` is not a public key or an asymmetric key pair, that is compatible with `alg`. • `hash_length` is not valid for the algorithm and key type. • `hash` is not a valid input value for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `alg` is not supported, or is not an asymmetric signature algorithm that permits verifying a pre-computed hash. • `key` is not supported for use with `alg`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

For hash-and-sign signature algorithms, the `hash` input to this function is the hash of the message to verify. The algorithm used to calculate this hash is encoded in the signature algorithm. For such algorithms, `hash_length` must equal the length of the hash output: `hash_length == PSA_HASH_LENGTH(PSA_ALG_GET_HASH(alg))`.

Specialized signature algorithms can apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. For example, see `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`.

If the algorithm has a context parameter, a zero-length context is used. To provide a context value, use `psa_verify_hash_with_context()` instead.

`psa_verify_hash_with_context` (function)

Verify the signature of a hash or short message using a public key and a supplied context.

Added in version 1.4.

```
psa_status_t psa_verify_hash_with_context(psa_key_id_t key,
                                          psa_algorithm_t alg,
                                          const uint8_t * hash,
                                          size_t hash_length,
                                          const uint8_t * context,
                                          size_t context_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 permit the usage `PSA_KEY_USAGE_VERIFY_HASH`.
`alg`	An asymmetric signature algorithm that separates the hash and sign operations: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_SIGN_HASH(alg)` is true.
`hash`	The input whose signature is to be verified. This is usually the hash of a message. See the description of this function, or the description of individual signature algorithms, for details of the acceptable inputs.
`hash_length`	Size of the `hash` buffer in bytes.
`context`	The context to use for this signature.
`context_length`	Size of the `context` 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`	Success. The signature is valid.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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`	`signature` is not the result of signing `hash` with algorithm `alg` using the private key corresponding to `key`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric signature algorithm that permits verifying a pre-computed hash with a context. • `key` is not a public key or an asymmetric key pair, that is compatible with `alg`.

- `hash_length` is not valid for the algorithm and key type.
- `hash` is not a valid input value for the algorithm and key type.
- `context_length` is not valid for the algorithm and key type.
- `context` is not a valid input value for the algorithm and key type.

`PSA_ERROR_NOT_SUPPORTED` The following conditions can result in this error:

- `alg` is not supported, or is not an asymmetric signature algorithm that permits verifying a pre-computed hash.
- The implementation does not support this value of `context_length` for `alg`.
- `key` is not supported for use with `alg`.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

For hash-and-sign signature algorithms, the `hash` input to this function is the hash of the message to verify. The algorithm used to calculate this hash is encoded in the signature algorithm. For such algorithms, `hash_length` must equal the length of the hash output: `hash_length == PSA_HASH_LENGTH(PSA_ALG_GET_HASH(alg))`.

Specialized signature algorithms can apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. For example, see `PSA_ALG_RSA_PKCS1V15_SIGN_RAW`.

If a context parameter is not required, `psa_verify_hash()` can be used instead.

10.9.5 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: a 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 a pre-computed 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.

Description

This macro evaluates to `1` for hash-and-sign and message-signature algorithms.

`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: a 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.

Description

This macro evaluates to `1` for hash-and-sign and specialized signature algorithms.

`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: a 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.

A wildcard signature algorithm policy, using `PSA_ALG_ANY_HASH`, returns the same value as the signature algorithm parameterized with a valid hash algorithm.

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. For example, if `PSA_ALG_IS_HASH_AND_SIGN(alg)` is true, the following call sequence is equivalent to `psa_sign_message(key, alg, msg, msg_len, ...)`:

```
uint8_t hash[PSA_HASH_MAX_SIZE];
size_t hash_len;
psa_hash_compute(PSA_ALG_GET_HASH(alg), msg, msg_len,
                 hash, sizeof(hash), &hash_len);
psa_sign_hash(key, alg, hash, hash_len, ...);
```

PSA_ALG_SIGN_SUPPORTS_CONTEXT (macro)

Whether the specified signature algorithm can be used with a non-zero-length context.

Added in version 1.4.

```
#define PSA_ALG_SIGN_SUPPORTS_CONTEXT(alg) /* implementation-defined value */
```

Parameters

alg	A signature algorithm identifier: a value of type psa_algorithm_t such that PSA_ALG_IS_SIGN(alg) is true.

Returns

1 if alg is a signature algorithm that can be used with a non-zero-length context. 0 if alg is a signature algorithm that cannot be used with a non-zero-length context. This macro can return either 0 or 1 if alg is not a supported signature algorithm identifier.

A wildcard signature algorithm policy, using PSA_ALG_ANY_HASH, returns the same value as the signature algorithm parameterized with a valid hash algorithm.

Description

This macro identifies signature algorithms that have a context parameter, and can be used with the appropriate functions that support non-zero-length contexts.

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 cannot be used to build an algorithm specification to perform an operation. If used in this way, the operation will fail with an error.

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_RSA_PSS_ANY_SALT
- 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

A sufficient signature buffer size for the specified asymmetric signature algorithm and key parameters. An implementation can return either 0 or a correct size for an asymmetric signature algorithm and key parameters that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the signature buffer is at least this large, it is guaranteed that psa_sign_message() and psa_sign_hash() 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_SIGNATURE_MAX_SIZE.

PSA_SIGNATURE_MAX_SIZE (macro)

A sufficient signature buffer size for psa_sign_message() and psa_sign_hash(), for any of the supported key types and asymmetric signature algorithms.

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

If the size of the signature buffer is at least this large, it is guaranteed that psa_sign_message() and psa_sign_hash() will not fail due to an insufficient buffer size.

See also PSA_SIGN_OUTPUT_SIZE().

10.10 Asymmetric encryption

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

10.10.1 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.

Compatible key types

`PSA_KEY_TYPE_RSA_KEY_PAIR`

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (asymmetric encryption only)

`PSA_ALG_RSA_OAEP` (macro)

The RSA OAEP asymmetric encryption algorithm.

```
#define PSA_ALG_RSA_OAEP(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true. The hash algorithm is used 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 following options:

- The mask generation function *MGF1* defined in [RFC8017] Appendix B.2.1.
- The specified hash algorithm is used to hash the label, and for the mask generation function.

Compatible key types

`PSA_KEY_TYPE_RSA_KEY_PAIR`

`PSA_KEY_TYPE_RSA_PUBLIC_KEY` (asymmetric encryption only)

10.10.2 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 permit the usage `PSA_KEY_USAGE_ENCRYPT`.
`alg`	The asymmetric encryption algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg)` is true.
`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`	Success. The first `(*output_length)` bytes of `output` contain the encrypted output.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 a sufficient buffer size.
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error: • `alg` is not an asymmetric encryption algorithm. • `key` is not a public key or an asymmetric key pair, that is compatible with `alg`. • `input_length` is not valid for the algorithm and key type. • `salt_length` is not valid for the algorithm and key type.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error: • `alg` is not supported or is not an asymmetric encryption algorithm. • `key` is not supported for use with `alg`. • `input_length` or `salt_length` are too large for the implementation.
PSA_ERROR_INSUFFICIENT_ENTROPY	
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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 permit the usage `PSA_KEY_USAGE_DECRYPT`.
`alg`	The asymmetric encryption algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg)` is true.
`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`	Success. The first `(*output_length)` bytes of `output` contain the decrypted output.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`key` is not a valid key identifier.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_PADDING`	The algorithm uses padding, and the input does not contain valid padding.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not an asymmetric encryption algorithm. • `key` is not an asymmetric key pair, that is compatible with `alg`. • `input_length` is not valid for the algorithm and key type. • `salt_length` is not valid for the algorithm and key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error:

- `alg` is not supported or is not an asymmetric encryption algorithm.
- `key` is not supported for use with `alg`.
- `input_length` or `salt_length` are too large for the implementation.

`PSA_ERROR_INSUFFICIENT_ENTROPY`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

- For `PSA_ALG_RSA_PKCS1V15_CRYPT`, no salt is supported.

10.10.3 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: a 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`	An asymmetric encryption algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg)` is true.

Returns

A sufficient output buffer size for the specified asymmetric encryption algorithm and key parameters. An implementation can return either `0` or a correct size for an asymmetric encryption algorithm and key parameters that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_asymmetric_encrypt()` 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_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE`.

`PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE` (macro)

A sufficient output buffer size for `psa_asymmetric_encrypt()`, for any of the supported key types and asymmetric encryption algorithms.

```
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
```

If the size of the output buffer is at least this large, it is guaranteed that `psa_asymmetric_encrypt()` will not fail due to an insufficient buffer size.

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`	An asymmetric encryption algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg)` is true.

Returns

A sufficient output buffer size for the specified asymmetric encryption algorithm and key parameters. An implementation can return either `0` or a correct size for an asymmetric encryption algorithm and key parameters that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_asymmetric_decrypt()` 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_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE`.

PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE (macro)

A sufficient output buffer size for psa_asymmetric_decrypt(), for any of the supported key types and asymmetric encryption algorithms.

```
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
```

If the size of the output buffer is at least this large, it is guaranteed that psa_asymmetric_decrypt() will not fail due to an insufficient buffer size.

See also PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE().

10.11 Key agreement

Three functions are provided for a Diffie-Hellman-style key agreement where each party combines its own private key with the peer's public key, to produce a shared secret value:

- A call to psa_key_agreement() will compute the shared secret and store the result in a new derivation key.
- If the resulting shared secret will be used for a single key derivation, a key-derivation operation can be used with the psa_key_derivation_key_agreement() input function. This calculates the shared secret and inputs it directly to the key-derivation operation.
- Where an application needs direct access to the shared secret, it can call psa_raw_key_agreement() instead.

Using psa_key_agreement() or psa_key_derivation_key_agreement() is recommended, as these do not expose the shared secret to the application.

> **Note:**
>
> In general the shared secret is not directly suitable for use as a key because it is biased.

10.11.1 Key-agreement algorithms

PSA_ALG_FFDH (macro)

The finite field Diffie-Hellman (DH) key-agreement algorithm.

```
#define PSA_ALG_FFDH ((psa_algorithm_t)0x09010000)
```

This standalone key-agreement algorithm can be used directly in a call to psa_key_agreement() or 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 a key's permitted-algorithm policy, the following uses are permitted:

- In a call to psa_key_agreement() or psa_raw_key_agreement(), with algorithm PSA_ALG_FFDH.
- In a call to psa_key_derivation_key_agreement(), with any combined key-agreement and key-derivation algorithm constructed with PSA_ALG_FFDH.

When used as part of a multi-part key-derivation operation, this implements a Diffie-Hellman key-agreement scheme using a single finite field 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 $\lceil (m/8) \rceil$ 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.

Compatible key types

`PSA_KEY_TYPE_DH_KEY_PAIR()`

`PSA_ALG_ECDH` (macro)

The elliptic curve Diffie-Hellman (ECDH) key-agreement algorithm.

```
#define PSA_ALG_ECDH ((psa_algorithm_t)0x09020000)
```

This standalone key-agreement algorithm can be used directly in a call to `psa_key_agreement()` or `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 a key's permitted-algorithm policy, the following uses are permitted:

- In a call to `psa_key_agreement()` or `psa_raw_key_agreement()`, with algorithm `PSA_ALG_ECDH`.
- In a call to `psa_key_derivation_key_agreement()`, with any combined key-agreement and key-derivation algorithm constructed with `PSA_ALG_ECDH`.

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 $\lceil (m/8) \rceil$ 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 $Z = hd_A Q_B = hd_B Q_A$ 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 $m = \lceil \log_2(p) \rceil$ for the field $\mathbb{F}_p$.
- Over binary fields (curve families `PSA_ECC_FAMILY_SECT_XX`), the bit size is m for the field $\mathbb{F}_{2^m}$.

> **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.

Compatible key types

`PSA_KEY_TYPE_ECC_KEY_PAIR(family)`

where `family` is a Weierstrass or Montgomery Elliptic curve family. That is, one of the following values:

- `PSA_ECC_FAMILY_SECT_XX`
- `PSA_ECC_FAMILY_SECP_XX`
- `PSA_ECC_FAMILY_FRP`
- `PSA_ECC_FAMILY_BRAINPOOL_P_R1`
- `PSA_ECC_FAMILY_MONTGOMERY`

`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: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_AGREEMENT(ka_alg)` is true.
`kdf_alg`	A key-derivation algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_DERIVATION(kdf_alg)` is true.

Returns

The corresponding key-agreement and key-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()`.

Compatible key types

The resulting combined key-agreement algorithm is compatible with the same key types as the standalone key-agreement algorithm used to construct it.

10.11.2 Standalone key agreement

`psa_key_agreement` (function)

Perform a key agreement and return the shared secret as a derivation key.

Added in version 1.2.

```
psa_status_t psa_key_agreement(psa_key_id_t private_key,
                               const uint8_t * peer_key,
                               size_t peer_key_length,
                               psa_algorithm_t alg,
                               const psa_key_attributes_t * attributes,
                               psa_key_id_t * key);
```

Parameters

`private_key`
: Identifier of the private key to use. It must permit the usage `PSA_KEY_USAGE_DERIVE`.

`peer_key`
: Public key of the peer. The peer key data is parsed with the type `PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type)` where `type` is the type of `private_key`, and with the same bit-size as `private_key`. The peer key must be in the format that `psa_import_key()` accepts for this public-key type. These formats are described with the public-key type in *Key types* on page 53.

`peer_key_length`
: Size of `peer_key` in bytes.

`alg`
: The standalone key-agreement algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg)` is true.

`attributes`
: The attributes for the new key.

  The following attributes are required for all keys:

  - The key type, which must be one of `PSA_KEY_TYPE_DERIVE`, `PSA_KEY_TYPE_RAW_DATA`, `PSA_KEY_TYPE_HMAC`, or `PSA_KEY_TYPE_PASSWORD`.
    Implementations must support the `PSA_KEY_TYPE_DERIVE` and `PSA_KEY_TYPE_RAW_DATA` key types.

  The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

The following attributes are optional:

- If the key size is nonzero, it must be equal to the output size of the key agreement, in bits.
  The output size, in bits, of the key agreement is `8 * PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(type, bits)`, where `type` and `bits` are the type and bit-size of `private_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. The new key contains the share secret. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`private_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	The following conditions can result in this error: • `private_key` does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the requested algorithm. • The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
`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 following conditions can result in this error: • `alg` is not a key-agreement algorithm. • `private_key` is not compatible with `alg`. • `peer_key` is not a valid public key corresponding to `private_key`. • The output key attributes in `attributes` are not valid : — The key type is not valid for key-agreement output. — The key size is nonzero, and is not the size of the shared

secret.
  - The key lifetime is invalid.
  - The key identifier is not valid for the key lifetime.
  - The key usage flags include invalid values.
  - The key's permitted-usage algorithm is invalid.
  - The key attributes, as a whole, are invalid.

`PSA_ERROR_NOT_SUPPORTED` The following conditions can result in this error:

- `alg` is not supported or is not a key-agreement algorithm.
- `private_key` is not supported for use with `alg`.
- The output key attributes, as a whole, are not supported, either by the implementation in general or in the specified storage location.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_INSUFFICIENT_STORAGE`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

A key-agreement algorithm takes two inputs: a private key `private_key`, and a public key `peer_key`. The result of this function is a shared secret, returned as a derivation key.

The new key's location, policy, and type are taken from `attributes`.

The size of the returned key is always the bit-size of the shared secret, rounded up to a whole number of bytes.

This key can be used as input to a key-derivation operation using `psa_key_derivation_input_key()`.

> **Warning**
>
> The shared secret resulting from a key-agreement algorithm such as finite field Diffie-Hellman or elliptic curve Diffie-Hellman has biases. This makes it unsuitable for use as key material, for example, as an AES key. Instead, it is recommended that a key-derivation algorithm is applied to the result, to derive unbiased cryptographic keys.

`psa_raw_key_agreement` (function)

Perform a key agreement and return the shared secret.

```
psa_status_t psa_raw_key_agreement(psa_algorithm_t alg,
                                   psa_key_id_t private_key,
                                   const uint8_t * peer_key,
```

(continues on next page)

(continued from previous page)

```
                                size_t peer_key_length,
                                uint8_t * output,
                                size_t output_size,
                                size_t * output_length);
```

Parameters

`alg`	The standalone key-agreement algorithm to compute: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg)` is true.
`private_key`	Identifier of the private key to use. It must permit the usage `PSA_KEY_USAGE_DERIVE`.
`peer_key`	Public key of the peer. The peer key data is parsed with the type `PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type)` where `type` is the type of `private_key`, and with the same bit-size as `private_key`. The peer key must be in the format that `psa_import_key()` accepts for this public-key type. These formats are described with the public-key type in *Key types* on page 53.
`peer_key_length`	Size of `peer_key` in bytes.
`output`	Buffer where the 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` and `bits` are the type and bit-size of `private_key`. • `PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported standalone key-agreement algorithm.
`output_length`	On success, the number of bytes that make up the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `(*output_length)` bytes of `output` contain the shared secret.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`private_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	`private_key` does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the requested algorithm.
`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 a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `alg` is not a key-agreement algorithm. • `private_key` is not compatible with `alg`.

- `peer_key` is not a valid public key corresponding to `private_key`.

`PSA_ERROR_NOT_SUPPORTED`

The following conditions can result in this error:

- `alg` is not supported or is not a key-agreement algorithm.
- `private_key` is not supported for use with `alg`.

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

A key-agreement algorithm takes two inputs: a private key `private_key`, and a public key `peer_key`. The result of this function is a shared secret, returned in the `output` buffer.

> **Warning**
>
> The result of a key-agreement algorithm such as finite field Diffie-Hellman or elliptic curve Diffie-Hellman has biases, and is not suitable for direct use as key material, for example, as an AES key. 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, either use `psa_key_agreement()` to obtain the result of the key agreement as a derivation key, or use `psa_key_derivation_key_agreement()` and other functions from the key-derivation interface.

10.11.3 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 combined key-agreement and key-derivation algorithm `alg`: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_AGREEMENT(alg)` is true and `PSA_ALG_IS_STANDALONE_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 permit the usage `PSA_KEY_USAGE_DERIVE`.
`peer_key`	Public key of the peer. The peer key data is parsed with the type `PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type)` where `type` is the type of `private_key`, and with the same bit-size as `private_key`. The peer key must be in the format that `psa_import_key()` accepts for this public-key type. These formats are described with the public-key type in *Key types* on page 53.
`peer_key_length`	Size of `peer_key` in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid for this key-agreement `step`. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`private_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	`private_key` does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the operation's algorithm.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The operation's algorithm is not a key-agreement algorithm. • `step` does not permit an input resulting from a key agreement. • `private_key` is not compatible with the operation's algorithm. • `peer_key` is not a valid public key corresponding to `private_key`.
`PSA_ERROR_NOT_SUPPORTED`	`private_key` is not supported for use with the operation's algorithm.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

A key-agreement algorithm takes two inputs: a private key `private_key`, and a public key `peer_key`. The result of this function is a shared secret, which is directly input to the key-derivation operation. Output from the key-derivation operation can then be used as 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()`.

Note:

This function cannot be used when the resulting shared secret is required for multiple key derivations.

Instead, the application can call psa_key_agreement() to obtain the shared secret as a derivation key. This key can be used as input to as many key-derivation operations as required.

10.11.4 Support macros

PSA_ALG_KEY_AGREEMENT_GET_BASE (macro)

Get the standalone key-agreement algorithm from a combined key-agreement and key-derivation algorithm.

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

Parameters

alg	A key-agreement algorithm: a value of type psa_algorithm_t such that PSA_ALG_IS_KEY_AGREEMENT(alg) is true.

Returns

The underlying standalone 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 combined key-agreement and key-derivation algorithm.

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

Parameters

alg	A key-agreement algorithm: a 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_STANDALONE_KEY_AGREEMENT (macro)

Whether the specified algorithm is a standalone key-agreement algorithm.

Added in version 1.2.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

Description

A standalone key-agreement algorithm is one that does not specify a key-derivation function. Usually, standalone key-agreement algorithms are constructed directly with a `PSA_ALG_xxx` macro while combined key-agreement algorithms are constructed with `PSA_ALG_KEY_AGREEMENT()`.

The standalone key-agreement algorithm can be extracted from a combined key-agreement algorithm identifier using `PSA_ALG_KEY_AGREEMENT_GET_BASE()`.

`PSA_ALG_IS_RAW_KEY_AGREEMENT` (macro)

Whether the specified algorithm is a standalone key-agreement algorithm.

Deprecated since version 1.2: Use `PSA_ALG_IS_STANDALONE_KEY_AGREEMENT()` instead.

```
#define PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) \
    PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg)
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

`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: a 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 standalone finite field Diffie-Hellman algorithm, as well as finite field Diffie-Hellman combined with any supported 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: a 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 standalone elliptic curve Diffie-Hellman algorithm, as well as elliptic curve Diffie-Hellman combined with any supported 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

A sufficient output buffer size for the specified key type and size. An implementation can return either `0` or a correct size for a key type and size that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_raw_key_agreement()` 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_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE`.

`PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE` (macro)

Sufficient output buffer size for `psa_raw_key_agreement()`, for any of the supported key types and key-agreement algorithms.

```
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
```

If the size of the output buffer is at least this large, it is guaranteed that `psa_raw_key_agreement()` will not fail due to an insufficient buffer size.

See also `PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE()`.

10.12 Key encapsulation

A key-encapsulation algorithm can be used by two participants to establish a shared secret key over a public channel. The shared secret key can then be used with symmetric-key cryptographic algorithms. Key-encapsulation algorithms are often referred to as 'key-encapsulation mechanisms' or KEMs.

In a key-encapsulation algorithm, participants A and B establish a shared secret as follows:

1. Participant A generates a key pair: a private decapsulation key, and a public encapsulation key.
2. The public encapsulation key is made available to participant B.
3. Participant B uses the encapsulation key to generate one copy of a shared secret, and some ciphertext.
4. The ciphertext is transferred to participant A.
5. Participant A uses the private decapsulation key to compute another copy of the shared secret.

Typically, the shared secret is used as input to a key-derivation function, to create keys for secure communication between participants A and B. However, some key-encapsulation algorithms result in a uniformly pseudorandom shared secret, which is suitable to be used directly as a cryptographic key.

Applications can use the resulting keys for different use cases. For example:

- Encrypting and authenticating a single non-interactive message from participant B to participant A.
- Securing an interactive communication channel between participants A and B.

10.12.1 Elliptic Curve Integrated Encryption Scheme

The Elliptic Curve Integrated Encryption Scheme (ECIES) was first proposed by Shoup, then improved by Ballare and Rogaway.

The original specification permitted a number of variants. The Crypto API uses the version specified in *SEC 1: Elliptic Curve Cryptography* [SEC1].

The full ECIES scheme uses an elliptic-curve key agreement between the recipient's static public key and an ephemeral private key, to establish encryption and authentication keys for secure transmission of arbitrary-length messages to the recipient.

An application using ECIES must select all of the following parameters:

- The elliptic curve for the initial key agreement.
- The KDF to derive the symmetric keys, and any label used in that derivation.
- The encryption and MAC algorithms.
- The additional data to include when computing the authentication.

The Crypto API presents the key-agreement step of ECIES as a key-encapsulation algorithm. The key derivation, encryption, and authentication steps are left to the application.

> **Implementation note**
>
> It is possible that some applications may need to use alternative versions of ECIES to interoperate with legacy systems.

While the application can implement this using key agreement functions, an implementation can choose to add these as a convenience with an IMPLEMENTATION DEFINED key-encapsulation algorithm identifier.

PSA_ALG_ECIES_SEC1 (macro)

The Elliptic Curve Integrated Encryption Scheme (ECIES).

Added in version 1.3.

```
#define PSA_ALG_ECIES_SEC1 ((psa_algorithm_t)0x0c000100)
```

This key-encapsulation algorithm is defined by *SEC 1: Elliptic Curve Cryptography* [SEC1] §5.1 under the name Elliptic Curve Integrated Encryption Scheme.

A call to `psa_encapsulate()` carries out steps 1 to 4 of the ECIES encryption process described in [SEC1] §5.1.3:

- The elliptic curve to use is determined by the key.
- The public-key part of the input key is used as Q_V.
- Cofactor ECDH is used to perform the key agreement.
- The octet string Z is output as the shared secret key.
- The ephemeral public key $\overline{R}$ is output as the ciphertext.

A call to `psa_decapsulate()` carries out steps 2 to 5 of the ECIES decryption process described in [SEC1] §5.1.4:

- The elliptic curve to use is determined by the key.
- The ciphertext is decoded as $\overline{R}$.
- The private key of the input key is used as d_V.
- Cofactor ECDH is used to perform the key agreement.
- The octet string Z is output as the shared secret key.

The ciphertext produced by `PSA_ALG_ECIES_SEC1` is not authenticated. In the full ECIES scheme, the authentication of the encrypted message using a key derived from the shared secret provides assurance that the message has not been manipulated.

The shared secret key that is produced by `PSA_ALG_ECIES_SEC1` is not suitable for use as an encryption key. It must be used as an input to a key derivation operation to produce additional cryptographic keys.

Compatible key types

`PSA_KEY_TYPE_ECC_KEY_PAIR(family)`

`PSA_KEY_TYPE_ECC_PUBLIC_KEY(family)` (encapsulaton only)

where `family` is a Weierstrass or Montgomery Elliptic curve family. That is, one of the following values:

- `PSA_ECC_FAMILY_SECT_XX`

- PSA_ECC_FAMILY_SECP_XX
- PSA_ECC_FAMILY_FRP
- PSA_ECC_FAMILY_BRAINPOOL_P_R1
- PSA_ECC_FAMILY_MONTGOMERY

10.12.2 Key-encapsulation functions

psa_encapsulate (function)

Use a public key to generate a new shared secret key and associated ciphertext.

Added in version 1.3.

```
psa_status_t psa_encapsulate(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const psa_key_attributes_t * attributes,
                             psa_key_id_t * output_key,
                             uint8_t * ciphertext,
                             size_t ciphertext_size,
                             size_t * ciphertext_length);
```

Parameters

`key`
: Identifier of the key to use for the encapsulation. It must be a public key or an asymmetric key pair. It must permit the usage PSA_KEY_USAGE_ENCRYPT.

`alg`
: The key-encapsulation algorithm to use: a value of type psa_algorithm_t such that PSA_ALG_IS_KEY_ENCAPSULATION(alg) is true.

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

  - The key type. All key-encapsulation algorithms can output a key of type PSA_KEY_TYPE_DERIVE or PSA_KEY_TYPE_HMAC. Key-encapsulation algorithms that produce a uniformly pseudorandom shared secret, can also output block-cipher key types, for example PSA_KEY_TYPE_AES. Refer to the documentation of individual key-encapsulation algorithms for more information.

  The following attributes must be set for keys used in cryptographic operations:

  - The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
  - The key usage flags, see *Key usage flags* on page 102.

  The following attributes must be set for keys that do not use the default PSA_KEY_LIFETIME_VOLATILE lifetime:

  - The key lifetime, see *Key lifetimes* on page 90.
  - The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

  The following attributes are optional:

- If the key size is nonzero, it must be equal to the size, in bits, of the shared secret.

> **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.

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

`ciphertext` — Buffer where the ciphertext output 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 ciphertext size is `PSA_ENCAPSULATE_CIPHERTEXT_SIZE(type, bits, alg)`, where `type` and `bits` are the type and bit-size of `key`.
- `PSA_ENCAPSULATE_CIPHERTEXT_MAX_SIZE` evaluates to the maximum ciphertext size of any supported key-encapsulation algorithm.

`ciphertext_length` — On success, the number of bytes that make up the ciphertext value.

Returns: `psa_status_t`

`PSA_SUCCESS` — Success. The bytes of `ciphertext` contain the data to be sent to the other participant, and `output_key` contains the identifier for the shared secret key.

`PSA_ERROR_BAD_STATE` — The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_HANDLE` — `key` is not a valid key identifier.

`PSA_ERROR_NOT_PERMITTED` — The following conditions can result in this error:

- `key` does not have the `PSA_KEY_USAGE_ENCRYPT` flag, or it does not permit the requested algorithm.
- The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.

`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_BUFFER_TOO_SMALL` — The size of the `ciphertext` buffer is too small. `PSA_ENCAPSULATE_CIPHERTEXT_SIZE()` or `PSA_ENCAPSULATE_CIPHERTEXT_MAX_SIZE` can be used to determine a sufficient buffer size.

`PSA_ERROR_INVALID_ARGUMENT` — The following conditions can result in this error:

- `alg` is not a key-encapsulation algorithm.
- `key` is not a public key or an asymmetric key pair, that is compatible with `alg`.
- The output key attributes in `attributes` are not valid:
  - The key type is not valid for the shared secret.

— The key size is nonzero, and is not the size of the shared secret.
— The key lifetime is invalid.
— The key identifier is not valid for the key lifetime.
— The key usage flags include invalid values.
— The key's permitted-usage algorithm is invalid.
— The key attributes, as a whole, are invalid.

`PSA_ERROR_NOT_SUPPORTED` The following conditions can result in this error:

- `alg` is not supported or is not a key-encapsulation algorithm.
- `key` is not supported for use with `alg`.
- The output key attributes in `attributes`, as a whole, are not supported, either by the implementation in general or in the specified storage location.

`PSA_ERROR_INSUFFICIENT_ENTROPY`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_INSUFFICIENT_STORAGE`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

The `output_key` location, policy, and type are taken from `attributes`.

The size of the returned key is always the bit-size of the shared secret, rounded up to a whole number of bytes. The size of the shared secret is dependent on the key-encapsulation algorithm and the type and size of `key`.

It is recommended that the shared secret key is used as an input to a key derivation operation to produce additional cryptographic keys. For some key-encapsulation algorithms, the shared secret key is also suitable for use as a key in cryptographic operations such as encryption. Refer to the documentation of individual key-encapsulation algorithms for more information.

The output `ciphertext` is to be sent to the other participant, who uses the decapsulation key to extract another copy of the shared secret key.

`psa_decapsulate` (function)

Use a private key to decapsulate a shared secret key from a ciphertext.

Added in version 1.3.

```
psa_status_t psa_decapsulate(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const uint8_t * ciphertext,
```

(continues on next page)

(continued from previous page)

```
                                size_t ciphertext_length,
                                const psa_key_attributes_t * attributes,
                                psa_key_id_t * output_key);
```

Parameters

`key`
: Identifier of the key to use for the decapsulation. It must be an asymmetric key pair. It must permit the usage `PSA_KEY_USAGE_DECRYPT`.

`alg`
: The key-encapsulation algorithm to use: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_ENCAPSULATION(alg)` is true.

`ciphertext`
: The ciphertext received from the other participant.

`ciphertext_length`
: Size of the `ciphertext` buffer in bytes.

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

  - The key type. All key-encapsulation algorithms can output a key of type `PSA_KEY_TYPE_DERIVE` or `PSA_KEY_TYPE_HMAC`. Key-encapsulation algorithms that produce a uniformly pseudorandom shared secret, can also output block-cipher key types, for example `PSA_KEY_TYPE_AES`. Refer to the documentation of individual key-encapsulation algorithms for more information.

  The following attributes must be set for keys used in cryptographic operations:

  - The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
  - The key usage flags, see *Key usage flags* on page 102.

  The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

  - The key lifetime, see *Key lifetimes* on page 90.
  - The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

  The following attributes are optional:

  - If the key size is nonzero, it must be equal to the size, in bits, of the shared secret.

  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.

`output_key`
: On success, an identifier for the newly created shared secret key. `PSA_KEY_ID_NULL` on failure.

Returns: `psa_status_t`

`PSA_SUCCESS` — Success. `output_key` contains the identifier for the shared secret key.

> **Note:**
> In some key-encapsulation algorithms, decapsulation failure is not reported with a explicit error code. Instead, an incorrect, pseudorandom key is output.

`PSA_ERROR_BAD_STATE` — The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_HANDLE` — `key` is not a valid key identifier.

`PSA_ERROR_NOT_PERMITTED` — The following conditions can result in this error:

- `key` does not have the `PSA_KEY_USAGE_DECRYPT` flag, or it does not permit the requested algorithm.
- The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.

`PSA_ERROR_INVALID_SIGNATURE` — Authentication of the ciphertext fails.

> **Note:**
> Some key-encapsulation algorithms do not report an authentication failure explicitly. Instead, an incorrect, pseudorandom key is output.

`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 following conditions can result in this error:

- `alg` is not a key-encapsulation algorithm.
- `key` is not an asymmetric key pair, that is compatible with `alg`.
- The output key attributes in `attributes` are not valid:
  - The key type is not valid for the shared secret.
  - The key size is nonzero, and is not the size of the shared secret.
  - The key lifetime is invalid.
  - The key identifier is not valid for the key lifetime.
  - The key usage flags include invalid values.
  - The key's permitted-usage algorithm is invalid.
  - The key attributes, as a whole, are invalid.
- `ciphertext` is obviously invalid for the selected algorithm and key. For example, the implementation can detect that it has an incorrect length.

`PSA_ERROR_NOT_SUPPORTED` — The following conditions can result in this error:

- `alg` is not supported or is not a key-encapsulation algorithm.
- `key` is not supported for use with `alg`.
- The output key attributes in `attributes`, as a whole, are not

supported, either by the implementation in general or in the specified storage location.

`PSA_ERROR_INSUFFICIENT_ENTROPY`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_INSUFFICIENT_STORAGE`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

The `output_key` location, policy, and type are taken from `attributes`.

The size of the returned key is always the bit-size of the shared secret, rounded up to a whole number of bytes. The size of the shared secret is dependent on the key-encapsulation algorithm and the type and size of `key`.

It is recommended that the shared secret key is used as an input to a key derivation operation to produce additional cryptographic keys. For some key-encapsulation algorithms, the shared secret key is also suitable for use as a key in cryptographic operations such as encryption. Refer to the documentation of individual key-encapsulation algorithms for more information.

If the key-encapsulation protocol is executed correctly then, with overwhelming probability, the two copies of the shared secret are identical. However, the protocol does not protect one participant against the other participant executing it incorrectly, or against a third party modifying data in transit.

Warning

A `PSA_SUCCESS` result from `psa_decapsulate()` does not guarantee that the output key is identical to the key produced by the call to `psa_encapsulate()`. For example, `PSA_SUCCESS` can be returned with a mismatched shared secret key value in the following situations:

- The key-encapsulation algorithm does not authenticate the ciphertext. Manipulated or corrupted ciphertext will not be detected during decapsulation.
- The key-encapsulation algorithm reports authentication failure implicitly, by returning a pseudorandom key value. This is done to prevent disclosing information to an attacker that has manipulated the ciphertext.
- The key-encapsulation algorithm is probablistic, and will *extremely* rarely result in non-identical key values.

It is strongly recommended that the application uses the output key in a way that will confirm that the shared secret keys are identical.

Implementation note

For key-encapsulation algorithms which involve data padding when computing the ciphertext, the decapsulation algorithm **must not** report a distinct error status if invalid padding is detected.

Instead, it is recommended that the decapsulation fails implicitly when invalid padding is detected, returning a pseudorandom key.

10.12.3 Support macros

`PSA_ENCAPSULATE_CIPHERTEXT_SIZE` (macro)

Sufficient ciphertext buffer size for `psa_encapsulate()`, in bytes.

Added in version 1.3.

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

Parameters

`key_type`	A key type that is compatible with algorithm `alg`.
`key_bits`	The size of the key in bits.
`alg`	A key-encapsulation algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_KEY_ENCAPSULATION(alg)` is true.

Returns

A sufficient ciphertext buffer size for the specified algorithm, key type, and size. An implementation can return either `0` or a correct size for an algorithm, key type, and size that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the ciphertext buffer is at least this large, it is guaranteed that `psa_encapsulate()` will not fail due to an insufficient buffer size. The actual size of the ciphertext might be smaller in any given call.

See also `PSA_ENCAPSULATE_CIPHERTEXT_MAX_SIZE`.

`PSA_ENCAPSULATE_CIPHERTEXT_MAX_SIZE` (macro)

Sufficient ciphertext buffer size for `psa_encapsulate()`, for any of the supported key types and key-encapsulation algorithms.

Added in version 1.3.

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

If the size of the ciphertext buffer is at least this large, it is guaranteed that `psa_encapsulate()` will not fail due to an insufficient buffer size.

See also `PSA_ENCAPSULATE_CIPHERTEXT_SIZE()`.

10.13 Password-authenticated key exchange (PAKE)

PAKE protocols provide an interactive method for two or more parties to establish cryptographic keys based on knowledge of a low entropy secret, such as a password.

These can provide strong security for communication from a weak password, because the password is not directly communicated as part of the key exchange.

This chapter is divided into the following sections:

- *Common API for PAKE* — the common interface elements, including the PAKE operation.
- *The J-PAKE protocol* on page 366 — the J-PAKE protocol, and the associated interface elements.
- *The SPAKE2+ protocol* on page 371 — the SPAKE2+ protocols, and the associated interface elements.
- *The WPA3-SAE protocol* on page 381 — the WPA3-SAE protocol, and the associated interface elements.

10.13.1 Common API for PAKE

This section defines all of the common interfaces used to carry out a PAKE protocol:

- *PAKE primitives*
- *PAKE cipher suites* on page 342
- *PAKE roles* on page 347
- *PAKE step types* on page 349
- *Multi-part PAKE operations* on page 352
- *PAKE support macros* on page 364

10.13.2 PAKE primitives

A PAKE algorithm specifies a sequence of interactions between the participants. Many PAKE algorithms are designed to allow different cryptographic primitives to be used for the key establishment operation, so long as all the participants are using the same underlying cryptography.

The cryptographic primitive for a PAKE operation is specified using a `psa_pake_primitive_t` value, which can be constructed using the `PSA_PAKE_PRIMITIVE()` macro, or can be provided as a numerical constant value.

A PAKE primitive is required when constructing a PAKE cipher-suite object, `psa_pake_cipher_suite_t`, which fully specifies the PAKE operation to be carried out.

`psa_pake_primitive_t` (typedef)

Encoding of the primitive associated with the PAKE.

Added in version 1.1.

```
typedef uint32_t psa_pake_primitive_t;
```

PAKE primitive values are constructed using `PSA_PAKE_PRIMITIVE()`.

Figure 2 on page 339 shows how the components of the primitive are encoded into a `psa_pake_primitive_t` value.

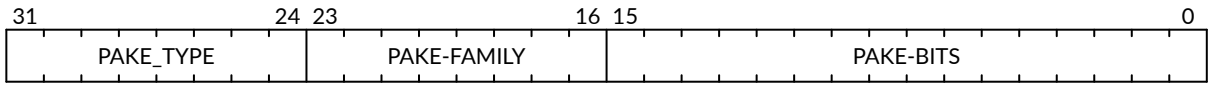

Figure 2 PAKE primitive encoding

The components of a PAKE primitive value can be extracted using the `PSA_PAKE_PRIMITIVE_GET_TYPE()`, `PSA_PAKE_PRIMITIVE_GET_FAMILY()`, and `PSA_PAKE_PRIMITIVE_GET_BITS()`. These can be used to set key attributes for keys used in PAKE algorithms. *SPAKE2+ registration* on page 372 provides an example of this usage.

`psa_pake_primitive_type_t` (typedef)

Encoding of the type of the PAKE's primitive.

Added in version 1.1.

```
typedef uint8_t psa_pake_primitive_type_t;
```

The range of PAKE primitive type values is divided as follows:

`0x00`
: Reserved as an invalid primitive type.

`0x01 - 0x7f`
: Specification-defined primitive type. Primitive types defined by this standard always have bit 7 clear. Unallocated primitive type values in this range are reserved for future use.

`0x80 - 0xff`
: Implementation-defined primitive type. Implementations that define additional primitive types must use an encoding with bit 7 set.

For specification-defined primitive types, see `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH`.

`PSA_PAKE_PRIMITIVE_TYPE_ECC` (macro)

The PAKE primitive type indicating the use of elliptic curves.

Added in version 1.1.

```
#define PSA_PAKE_PRIMITIVE_TYPE_ECC ((psa_pake_primitive_type_t)0x01)
```

The values of the `family` and `bits` components of the PAKE primitive identify a specific elliptic curve, using the same mapping that is used for ECC keys. See the definition of `psa_ecc_family_t`. Here `family` and `bits` refer to the values used to construct the PAKE primitive using `PSA_PAKE_PRIMITIVE()`.

Input and output during the operation can involve group elements and scalar values:

- The format for group elements is the same as that for public keys on the specific elliptic curve. See *Key format* within the definition of `PSA_KEY_TYPE_ECC_PUBLIC_KEY()`.
- The format for scalars is the same as that for private keys on the specific elliptic curve. See *Key format* within the definition of `PSA_KEY_TYPE_ECC_KEY_PAIR()`.

PSA_PAKE_PRIMITIVE_TYPE_DH (macro)

The PAKE primitive type indicating the use of a finite field Diffie-Hellman group.

Added in version 1.1.

```
#define PSA_PAKE_PRIMITIVE_TYPE_DH ((psa_pake_primitive_type_t)0x02)
```

The values of the `family` and `bits` components of the PAKE primitive identify a specific finite field Diffie-Hellman group, using the same mapping that is used for finite field Diffie-Hellman keys. See the definition of `psa_dh_family_t`. Here `family` and `bits` refer to the values used to construct the PAKE primitive using `PSA_PAKE_PRIMITIVE()`.

Input and output during the operation can involve group elements and scalar values:

- The format for group elements is the same as that for public keys in the specific finite field Diffie-Hellman group. See *Key format* within the definition of `PSA_KEY_TYPE_DH_PUBLIC_KEY()`.
- The format for scalars is the same as that for private keys in the specific finite field Diffie-Hellman group. See *Key format* within the definition of `PSA_KEY_TYPE_DH_PUBLIC_KEY()`.

psa_pake_family_t (typedef)

Encoding of the family of the primitive associated with the PAKE.

Added in version 1.1.

```
typedef uint8_t psa_pake_family_t;
```

For more information on the family values, see `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH`.

PSA_PAKE_PRIMITIVE (macro)

Construct a PAKE primitive from type, family and bit-size.

Added in version 1.1.

```
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
```

Parameters

`pake_type`	The type of the primitive: a value of type `psa_pake_primitive_type_t`.
`pake_family`	The family of the primitive. The type and interpretation of this parameter depends on `pake_type`. For more information, see `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH`.
`pake_bits`	The bit-size of the primitive: a value of type `size_t`. The interpretation of this parameter depends on `pake_type` and `family`. For more information, see `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH`.

Returns: `psa_pake_primitive_t`

The constructed primitive value. Return `0` if the requested primitive can't be encoded as `psa_pake_primitive_t`.

Description

A PAKE primitive value is used to specify a PAKE operation, as part of a PAKE cipher suite.

`PSA_PAKE_PRIMITIVE_GET_TYPE` (macro)

Extract the PAKE primitive type from a PAKE primitive.

Added in version 1.2.

```
#define PSA_PAKE_PRIMITIVE_GET_TYPE(pake_primitive) \
    /* specification-defined value */
```

Parameters

`pake_primitive`	A PAKE primitive: a value of type `psa_pake_primitive_t`.

Returns: `psa_pake_primitive_type_t`

The PAKE primitive type, if `pake_primitive` is a supported PAKE primitive. Unspecified if `pake_primitive` is not a supported PAKE primitive.

`PSA_PAKE_PRIMITIVE_GET_FAMILY` (macro)

Extract the family from a PAKE primitive.

Added in version 1.2.

```
#define PSA_PAKE_PRIMITIVE_GET_FAMILY(pake_primitive) \
    /* specification-defined value */
```

Parameters

`pake_primitive`	A PAKE primitive: a value of type `psa_pake_primitive_t`.

Returns: `psa_pake_family_t`

The PAKE primitive family, if `pake_primitive` is a supported PAKE primitive. Unspecified if `pake_primitive` is not a supported PAKE primitive.

Description

For more information on the family values, see `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH`.

`PSA_PAKE_PRIMITIVE_GET_BITS` (macro)

Extract the bit-size from a PAKE primitive.

Added in version 1.2.

```
#define PSA_PAKE_PRIMITIVE_GET_BITS(pake_primitive) \
    /* specification-defined value */
```

Parameters

`pake_primitive`	A PAKE primitive: a value of type `psa_pake_primitive_t`.

Returns: `size_t`

The PAKE primitive bit-size, if `pake_primitive` is a supported PAKE primitive. Unspecified if `pake_primitive` is not a supported PAKE primitive.

Description

For more information on the bit-size values, see `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH`.

10.13.3 PAKE cipher suites

Most PAKE algorithms have parameters that must be specified by the application. These parameters include the following:

- The cryptographic primitive used for key establishment, specified using a PAKE primitive.
- A cryptographic hash algorithm.
- Whether the application requires the shared secret before, or after, it is confirmed.

The hash algorithm is encoded into the PAKE algorithm identifier. The `psa_pake_cipher_suite_t` object is used to fully specify a PAKE operation, combining the PAKE protocol with all of the above parameters.

A PAKE cipher suite is required when setting up a PAKE operation in `psa_pake_setup()`.

`psa_pake_cipher_suite_t` (typedef)

The type of an object describing a PAKE cipher suite.

Added in version 1.1.

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

This is the object that represents the cipher suite used for a PAKE algorithm. The PAKE cipher suite specifies the PAKE algorithm, and the options selected for that algorithm. The cipher suite includes the following attributes:

- The PAKE algorithm itself.
- The hash algorithm, encoded within the PAKE algorithm.
- The PAKE primitive, which identifies the prime order group used for the key exchange operation. See *PAKE primitives* on page 338.
- Whether to confirm the shared secret.

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

Before calling any function on a PAKE cipher suite object, the application must initialize it by any of the following means:

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

```
psa_pake_cipher_suite_t cipher_suite;
memset(&cipher_suite, 0, sizeof(cipher_suite));
```

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

```
static psa_pake_cipher_suite_t cipher_suite;
```

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

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;
```

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

```
psa_pake_cipher_suite_t cipher_suite;
cipher_suite = psa_pake_cipher_suite_init();
```

Following initialization, the cipher-suite object contains the following values:

Attribute	Value
algorithm	`PSA_ALG_NONE` — an invalid algorithm identifier.
primitive	`0` — an invalid PAKE primitive.
key confirmation	`PSA_PAKE_CONFIRMED_KEY` — requesting that the secret key is confirmed before it can be returned.

Valid algorithm, primitive, and key confirmation values must be set when using a PAKE cipher suite.

> **Implementation note**
>
> Implementations are recommended to define the cipher-suite object as a simple data structure, with fields corresponding to the individual cipher suite attributes. In such an implementation, each function `psa_pake_cs_set_xxx()` sets a field and the corresponding function `psa_pake_cs_get_xxx()` retrieves the value of the field.
>
> An implementation 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 attributes 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.

`PSA_PAKE_CIPHER_SUITE_INIT` (macro)

This macro returns a suitable initializer for a PAKE cipher suite object of type `psa_pake_cipher_suite_t`.

Added in version 1.1.

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

`psa_pake_cipher_suite_init` (function)

Return an initial value for a PAKE cipher suite object.

Added in version 1.1.

```
psa_pake_cipher_suite_t psa_pake_cipher_suite_init(void);
```

Returns: `psa_pake_cipher_suite_t`

`psa_pake_cs_get_algorithm` (function)

Retrieve the PAKE algorithm from a PAKE cipher suite.

Added in version 1.1.

```
psa_algorithm_t psa_pake_cs_get_algorithm(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `psa_algorithm_t`

The PAKE algorithm stored in the cipher suite object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_algorithm` (function)

Declare the PAKE algorithm for the cipher suite.

Added in version 1.1.

```
void psa_pake_cs_set_algorithm(psa_pake_cipher_suite_t* cipher_suite,
                               psa_algorithm_t alg);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`alg`	The PAKE algorithm to write: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.

Returns: `void`

Description

This function overwrites any PAKE algorithm previously set in `cipher_suite`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_get_primitive` (function)

Retrieve the primitive from a PAKE cipher suite.

Added in version 1.1.

```
psa_pake_primitive_t psa_pake_cs_get_primitive(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `psa_pake_primitive_t`

The primitive stored in the cipher suite object.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_primitive` (function)

Declare the primitive for a PAKE cipher suite.

Added in version 1.1.

```
void psa_pake_cs_set_primitive(psa_pake_cipher_suite_t* cipher_suite,
                               psa_pake_primitive_t primitive);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`primitive`	The PAKE primitive to write: a value of type `psa_pake_primitive_t`. If this is `0`, the primitive type in `cipher_suite` becomes unspecified.

Returns: `void`

Description

This function overwrites any primitive previously set in `cipher_suite`.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`PSA_PAKE_CONFIRMED_KEY` (macro)

A key confirmation value that indicates an confirmed key in a PAKE cipher suite.

Added in version 1.2.

```
#define PSA_PAKE_CONFIRMED_KEY 0
```

This key confirmation value will result in the PAKE algorithm exchanging data to verify that the shared key is identical for both parties. This is the default key confirmation value in an initialized PAKE cipher suite object.

Some algorithms do not include confirmation of the shared key.

`PSA_PAKE_UNCONFIRMED_KEY` (macro)

A key confirmation value that indicates an unconfirmed key in a PAKE cipher suite.

Added in version 1.2.

```
#define PSA_PAKE_UNCONFIRMED_KEY 1
```

This key confirmation value will result in the PAKE algorithm terminating prior to confirming that the resulting shared key is identical for both parties.

Some algorithms do not support returning an unconfirmed shared key.

> **Warning**
>
> When the shared key is not confirmed as part of the PAKE operation, the application is responsible for mitigating risks that arise from the possible mismatch in the output keys.

`psa_pake_cs_get_key_confirmation` (function)

Retrieve the key confirmation from a PAKE cipher suite.

Added in version 1.2.

```
uint32_t psa_pake_cs_get_key_confirmation(const psa_pake_cipher_suite_t* cipher_suite);
```

Parameters

`cipher_suite`	The cipher suite object to query.

Returns: `uint32_t`

A key confirmation value: either `PSA_PAKE_CONFIRMED_KEY` or `PSA_PAKE_UNCONFIRMED_KEY`.

Description

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

`psa_pake_cs_set_key_confirmation` (function)

Declare the key confirmation from a PAKE cipher suite.

Added in version 1.2.

```
void psa_pake_cs_set_key_confirmation(psa_pake_cipher_suite_t* cipher_suite,
                                      uint32_t key_confirmation);
```

Parameters

`cipher_suite`	The cipher suite object to write to.
`key_confirmation`	The key confirmation value to write: either `PSA_PAKE_CONFIRMED_KEY` or `PSA_PAKE_UNCONFIRMED_KEY`.

Returns: `void`

Description

This function overwrites any key confirmation previously set in `cipher_suite`.

The documentation of individual PAKE algorithms specifies which key confirmation values are valid for the algorithm.

> **Implementation note**
>
> This is a simple accessor function that is not required to validate its inputs. It can be efficiently implemented as a `static inline` function or a function-like macro.

10.13.4 PAKE roles

Some PAKE algorithms need to know which role each participant is taking in the algorithm. For example:

- Augmented PAKE algorithms typically have a client and a server participant.
- Some symmetric PAKE algorithms assign an order to the two participants.

`psa_pake_role_t` (typedef)

Encoding of the application role in a PAKE algorithm.

Added in version 1.1.

```
typedef uint8_t psa_pake_role_t;
```

This type is used to encode the application's role in the algorithm being executed. For more information see the documentation of individual PAKE role constants.

`PSA_PAKE_ROLE_NONE` (macro)

A value to indicate no role in a PAKE algorithm.

Added in version 1.1.

```
#define PSA_PAKE_ROLE_NONE ((psa_pake_role_t)0x00)
```

This value can be used in a call to `psa_pake_set_role()` for symmetric PAKE algorithms which do not assign roles.

`PSA_PAKE_ROLE_FIRST` (macro)

The first peer in a balanced PAKE.

Added in version 1.1.

```
#define PSA_PAKE_ROLE_FIRST ((psa_pake_role_t)0x01)
```

Although balanced PAKE algorithms are symmetric, some of them need the peers to be ordered for the transcript calculations. If the algorithm does not need a specific ordering, then either do not call `psa_pake_set_role()`, or use `PSA_PAKE_ROLE_NONE` as the role parameter.

`PSA_PAKE_ROLE_SECOND` (macro)

The second peer in a balanced PAKE.

Added in version 1.1.

```
#define PSA_PAKE_ROLE_SECOND ((psa_pake_role_t)0x02)
```

Although balanced PAKE algorithms are symmetric, some of them need the peers to be ordered for the transcript calculations. If the algorithm does not need a specific ordering, then either do not call `psa_pake_set_role()`, or use `PSA_PAKE_ROLE_NONE` as the role parameter.

`PSA_PAKE_ROLE_CLIENT` (macro)

The client in an augmented PAKE.

Added in version 1.1.

```
#define PSA_PAKE_ROLE_CLIENT ((psa_pake_role_t)0x11)
```

Augmented PAKE algorithms need to differentiate between client and server.

PSA_PAKE_ROLE_SERVER (macro)

The server in an augmented PAKE.

Added in version 1.1.

```
#define PSA_PAKE_ROLE_SERVER ((psa_pake_role_t)0x12)
```

Augmented PAKE algorithms need to differentiate between client and server.

10.13.5 PAKE step types

psa_pake_step_t (typedef)

Encoding of input and output steps for a PAKE algorithm.

Added in version 1.1.

```
typedef uint8_t psa_pake_step_t;
```

Some PAKE algorithms need to exchange more data than a single key share. This type encodes additional input and output steps for such algorithms.

PSA_PAKE_STEP_KEY_SHARE (macro)

A key share being sent to or received from a PAKE participant.

Added in version 1.1.

```
#define PSA_PAKE_STEP_KEY_SHARE ((psa_pake_step_t)0x01)
```

The format for both input and output using this step is the same as the format for public keys on the group specified by the PAKE operation's primitive.

The public-key formats are defined in the documentation for psa_export_public_key().

For information regarding how the group is determined, consult the documentation PSA_PAKE_PRIMITIVE().

PSA_PAKE_STEP_ZK_PUBLIC (macro)

A Schnorr NIZKP public key being sent to or received from a PAKE participant.

Added in version 1.1.

```
#define PSA_PAKE_STEP_ZK_PUBLIC ((psa_pake_step_t)0x02)
```

This is the ephemeral public key in the Schnorr Non-Interactive Zero-Knowledge Proof, this is the value denoted by *V* in [RFC8235].

The format for both input and output at this step is the same as that for public keys on the group specified by the PAKE operation's primitive.

For more information on the format, consult the documentation of psa_export_public_key().

For information regarding how the group is determined, consult the documentation PSA_PAKE_PRIMITIVE().

`PSA_PAKE_STEP_ZK_PROOF` (macro)

A Schnorr NIZKP proof being sent to or received from a PAKE participant.

Added in version 1.1.

```
#define PSA_PAKE_STEP_ZK_PROOF ((psa_pake_step_t)0x03)
```

This is the proof in the Schnorr Non-Interactive Zero-Knowledge Proof, this is the value denoted by *r* in [RFC8235].

Both for input and output, the value at this step is an integer less than the order of the group specified by the PAKE operation's primitive. The format depends on the group as well:

- For Montgomery curves, the encoding is little endian.
- For other elliptic curves, and for finite field Diffie-Hellman groups, the encoding is big endian. See [SEC1] §2.3.8.

In both cases leading zeroes are permitted as long as the length in bytes does not exceed the byte length of the group order.

For information regarding how the group is determined, consult the documentation `PSA_PAKE_PRIMITIVE()`.

`PSA_PAKE_STEP_CONFIRM` (macro)

A key confirmation value being sent to or received from a PAKE participant.

Added in version 1.2.

```
#define PSA_PAKE_STEP_CONFIRM ((psa_pake_step_t)0x04)
```

This value is used during the key confirmation phase of a PAKE protocol. The use of this step, and format of the value depends on the algorithm and cipher suite:

- For a SPAKE2+ algorithm, the format for both input and output at this step is the same as the output of the MAC algorithm specified in the cipher suite. See *SPAKE2+ operation* on page 374.
- For a WPA3-SAE algorithm, the format for both input and output at this step is a 2-byte little-endian *send-confirm* counter, followed by the *confirm* value, which is the output from the hash algorithm specified in the cipher suite. See *WPA3-SAE operation* on page 384.

`PSA_PAKE_STEP_SALT` (macro)

A salt value used for deriving shared secrets within a PAKE operation.

Added in version 1.4.

```
#define PSA_PAKE_STEP_SALT ((psa_pake_step_t)0x05)
```

This input can be used during the key exchange phase of a PAKE protocol. The use of this step, and format of the value depends on the algorithm and cipher suite:

- For a WPA3-SAE algorithm, a salt value must be provided as defined in [IEEE-802.11] §12.4.5.4. See *WPA3-SAE operation* on page 384.

`PSA_PAKE_STEP_COMMIT` (macro)

A commitment value being sent to or received from a PAKE participant.

Added in version 1.4.

```
#define PSA_PAKE_STEP_COMMIT ((psa_pake_step_t)0x06)
```

This input and output is used during the key exchange phase of a PAKE protocol. The use of this step, and format of the value depends on the algorithm and cipher suite:

- For a WPA3-SAE algorithm, the format for input and output at this step is a concatenation of the *commit-scalar* and *COMMIT-ELEMENT* values, as defined in [IEEE-802.11] §12.4.7.3.
  See *WPA3-SAE operation* on page 384.

> **Note:**
> These values are adjacent in the WPA3-SAE Authentication frame defined in [IEEE-802.11] §9.3.3.11. The concatenated value can be output directly to, or input directly from, the frame buffer.

`PSA_PAKE_STEP_CONFIRM_COUNT` (macro)

A counter used as part of key confirmation.

Added in version 1.4.

```
#define PSA_PAKE_STEP_CONFIRM_COUNT ((psa_pake_step_t)0x07)
```

This value is input during the key confirmation phase of a PAKE protocol. It enables multiple confirmation attempts to result in distinct confirmation values. The use of this step, and format of the value depends on the algorithm and cipher suite:

- For a WPA3-SAE algorithm, the format for input at this step is the 2-byte little-endian *send-confirm* counter. See *WPA3-SAE operation* on page 384.

`PSA_PAKE_STEP_KEY_ID` (macro)

A key identifier value from a PAKE operation.

Added in version 1.4.

```
#define PSA_PAKE_STEP_KEY_ID ((psa_pake_step_t)0x08)
```

This value can be output from a PAKE operation following key confirmation. The use of this step, and format of the value depends on the algorithm and cipher suite:

- For a WPA3-SAE algorithm, the format of the output at this step is the 16-byte PMKID. See *WPA3-SAE operation* on page 384.

10.13.6 Multi-part PAKE operations

`psa_pake_operation_t` (typedef)

The type of the state object for PAKE operations.

Added in version 1.1.

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

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

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

  ```
  psa_pake_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_pake_operation_t operation;
  ```

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

  ```
  psa_pake_operation_t operation = PSA_PAKE_OPERATION_INIT;
  ```

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

  ```
  psa_pake_operation_t operation;
  operation = psa_pake_operation_init();
  ```

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

`PSA_PAKE_OPERATION_INIT` (macro)

This macro returns a suitable initializer for a PAKE operation object of type `psa_pake_operation_t`.

Added in version 1.1.

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

`psa_pake_operation_init` (function)

Return an initial value for a PAKE operation object.

Added in version 1.1.

```
psa_pake_operation_t psa_pake_operation_init(void);
```

Returns: `psa_pake_operation_t`

`psa_pake_setup` (function)

Setup a password-authenticated key exchange.

Added in version 1.1.

Changed in version 1.2: Added key to the operation setup.

```
psa_status_t psa_pake_setup(psa_pake_operation_t * operation,
                            psa_key_id_t password_key,
                            const psa_pake_cipher_suite_t * cipher_suite);
```

Parameters

`operation`	The operation object to set up. It must have been initialized as per the documentation for `psa_pake_operation_t` and not yet in use.
`password_key`	Identifier of the key holding the password or a value derived from the password. It must remain valid until the operation terminates. The valid key types depend on the PAKE algorithm, and participant role. Refer to the documentation of individual PAKE algorithms for more information. The key must permit the usage `PSA_KEY_USAGE_DERIVE`.
`cipher_suite`	The cipher suite to use. A PAKE cipher suite fully characterizes a PAKE algorithm, including the PAKE algorithm. The cipher suite must be compatible with the key type of `password_key`.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation is now active.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be inactive. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_HANDLE`	`password_key` is not a valid key identifier.
`PSA_ERROR_NOT_PERMITTED`	`password_key` does not have the `PSA_KEY_USAGE_DERIVE` flag, or it does not permit the algorithm in `cipher_suite`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • The algorithm in `cipher_suite` is not a PAKE algorithm, or encodes an invalid hash algorithm. • The PAKE primitive in `cipher_suite` is not compatible with the PAKE algorithm. • The key confirmation value in `cipher_suite` is not compatible with the PAKE algorithm and primitive. • The key type or key size of `password_key` is not compatible with `cipher_suite`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • The algorithm in `cipher_suite` is not a supported PAKE algorithm, or encodes an unsupported hash algorithm.

- The PAKE primitive in `cipher_suite` is not supported or not compatible with the PAKE algorithm.
- The key confirmation value in `cipher_suite` is not supported, or not compatible, with the PAKE algorithm and primitive.
- The key type or key size of `password_key` is not supported with `cipher suite`.

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

`PSA_ERROR_STORAGE_FAILURE`

`PSA_ERROR_DATA_CORRUPT`

`PSA_ERROR_DATA_INVALID`

Description

The sequence of operations to set up a password-authenticated key exchange operation is as follows:

1. Allocate a PAKE 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_pake_operation_t`. For example, using `PSA_PAKE_OPERATION_INIT`.
3. Call `psa_pake_setup()` to specify the cipher suite and provide the password or password-derived key.
4. Call `psa_pake_set_xxx()` functions on the operation to complete the setup. The exact sequence of `psa_pake_set_xxx()` functions that needs to be called depends on the algorithm in use.

A typical sequence of calls to perform a password-authenticated key exchange:

1. Call `psa_pake_output(operation, PSA_PAKE_STEP_KEY_SHARE, ...)` to get the key share that needs to be sent to the peer.
2. Call `psa_pake_input(operation, PSA_PAKE_STEP_KEY_SHARE, ...)` to provide the key share that was received from the peer.
3. Depending on the algorithm additional calls to `psa_pake_output()` and `psa_pake_input()` might be necessary.
4. Call `psa_pake_get_shared_key()` to access the shared secret.

Refer to the documentation of individual PAKE algorithms for details on the required set up and operation for each algorithm, and for constraints on the format and content of valid passwords.

After a successful call to `psa_pake_setup()`, the operation is active, and the application must eventually terminate the operation. The following events terminate an operation:

- A successful call to `psa_pake_get_shared_key()`.
- A call to `psa_pake_abort()`.

If `psa_pake_setup()` returns an error, the operation object is unchanged. If a subsequent function call with an active operation returns an error, the operation enters an error state.

To abandon an active operation, or reset an operation in an error state, call `psa_pake_abort()`.

See *Multi-part operations* on page 27.

`psa_pake_set_role` (function)

Set the application role for a password-authenticated key exchange.

Added in version 1.1.

```
psa_status_t psa_pake_set_role(psa_pake_operation_t * operation,
                               psa_pake_role_t role);
```

Parameters

`operation`	Active PAKE operation.
`role`	A value of type `psa_pake_role_t` indicating the application role in the PAKE algorithm. See *PAKE roles* on page 347.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_role()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `role` is not a valid PAKE role in the operation's algorithm. • `role` is not compatible with the operation's key type.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `role` is not a valid PAKE role, or is not supported for the operation's algorithm. • `role` is not supported with the operation's key type.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Not all PAKE algorithms need to differentiate the communicating participants. For PAKE algorithms that do not require a role to be specified, the application can do either of the following:

- Not call `psa_pake_set_role()` on the PAKE operation.
- Call `psa_pake_set_role()` with the `PSA_PAKE_ROLE_NONE` role.

Refer to the documentation of individual PAKE algorithms for more information.

`psa_pake_set_user` (function)

Set the user ID for a password-authenticated key exchange.

Added in version 1.1.

```
psa_status_t psa_pake_set_user(psa_pake_operation_t * operation,
                               const uint8_t * user_id,
                               size_t user_id_len);
```

Parameters

`operation`	Active PAKE operation.
`user_id`	The user ID to authenticate with.
`user_id_len`	Size of the `user_id` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_user()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`user_id` is not valid for the operation's algorithm and cipher suite.
`PSA_ERROR_NOT_SUPPORTED`	The value of `user_id` is not supported by the implementation.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Call this function to set the user ID. For PAKE algorithms that associate a user identifier with both participants in the session, also call `psa_pake_set_peer()` with the peer ID. For PAKE algorithms that associate a single user identifier with the session, call `psa_pake_set_user()` only.

Refer to the documentation of individual PAKE algorithms for more information.

`psa_pake_set_peer` (function)

Set the peer ID for a password-authenticated key exchange.

Added in version 1.1.

```
psa_status_t psa_pake_set_peer(psa_pake_operation_t * operation,
                               const uint8_t * peer_id,
                               size_t peer_id_len);
```

Parameters

`operation`	Active PAKE operation.
`peer_id`	The peer's ID to authenticate.
`peer_id_len`	Size of the `peer_id` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_peer()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet. • Calling `psa_pake_set_peer()` is invalid with the operation's algorithm. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_ARGUMENT`	`peer_id` is not valid for the operation's algorithm and cipher suite.
`PSA_ERROR_NOT_SUPPORTED`	The value of `peer_id` is not supported by the implementation.
`PSA_ERROR_NOT_SUPPORTED`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

Description

Call this function in addition to `psa_pake_set_user()` for PAKE algorithms that associate a user identifier with both participants in the session. For PAKE algorithms that associate a single user identifier with the session, call `psa_pake_set_user()` only.

Refer to the documentation of individual PAKE algorithms for more information.

`psa_pake_set_context` (function)

Set the context data for a password-authenticated key exchange.

Added in version 1.2.

```
psa_status_t psa_pake_set_context(psa_pake_operation_t * operation,
                                  const uint8_t * context,
                                  size_t context_len);
```

Parameters

`operation`	Active PAKE operation.
`context`	The peer's ID to authenticate.
`context_len`	Size of the `context` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active, and `psa_pake_set_context()`, `psa_pake_input()`, and `psa_pake_output()` must not have been called yet.

- Calling `psa_pake_set_context()` is invalid with the operation's algorithm.
- The library requires initializing by a call to `psa_crypto_init()`.

`PSA_ERROR_INVALID_ARGUMENT` `context` is not valid for the operation's algorithm and cipher suite.

`PSA_ERROR_NOT_SUPPORTED` The value of `context` is not supported by the implementation.

`PSA_ERROR_NOT_SUPPORTED`

`PSA_ERROR_INSUFFICIENT_MEMORY`

`PSA_ERROR_COMMUNICATION_FAILURE`

`PSA_ERROR_CORRUPTION_DETECTED`

Description

Call this function for PAKE algorithms that accept additional context data as part of the protocol setup.

Refer to the documentation of individual PAKE algorithms for more information.

`psa_pake_output` (function)

Get output for a step of a password-authenticated key exchange.

Added in version 1.1.

```
psa_status_t psa_pake_output(psa_pake_operation_t * operation,
                             psa_pake_step_t step,
                             uint8_t * output,
                             size_t output_size,
                             size_t * output_length);
```

Parameters

`operation` Active PAKE operation.

`step` The step of the algorithm for which the output is requested.

`output` Buffer where the output is to be written. The format of the output depends on the `step`, see *PAKE step types* on page 349.

`output_size` Size of the `output` buffer in bytes. This must be appropriate for the cipher suite and output step:

- A sufficient output size is `PSA_PAKE_OUTPUT_SIZE(alg, primitive, step)` where `alg` and `primitive` are the PAKE algorithm and primitive in the operation's cipher suite, and `step` is the output step.
- `PSA_PAKE_OUTPUT_MAX_SIZE` evaluates to the maximum output size of any supported PAKE algorithm, primitive and step.

`output_length` On success, the number of bytes of the returned output.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The first `(*output_length)` bytes of `output` contain the output.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active and fully set up, and this call must conform to the algorithm's requirements for ordering of input and output steps. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_BUFFER_TOO_SMALL`	The size of the `output` buffer is too small. `PSA_PAKE_OUTPUT_SIZE()` or `PSA_PAKE_OUTPUT_MAX_SIZE` can be used to determine a sufficient buffer size.
`PSA_ERROR_INVALID_ARGUMENT`	`step` is not compatible with the operation's algorithm.
`PSA_ERROR_NOT_SUPPORTED`	`step` is not supported with the operation's algorithm.
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

Depending on the algorithm being executed, you might need to call this function several times or you might not need to call this at all.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

`psa_pake_input` (function)

Provide input for a step of a password-authenticated key exchange.

Added in version 1.1.

```
psa_status_t psa_pake_input(psa_pake_operation_t * operation,
                            psa_pake_step_t step,
                            const uint8_t * input,
                            size_t input_length);
```

Parameters

`operation`	Active PAKE operation.
`step`	The step for which the input is provided.
`input`	Buffer containing the input. The format of the input depends on the `step`, see *PAKE step types* on page 349.
`input_length`	Size of the `input` buffer in bytes.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success.
`PSA_ERROR_BAD_STATE`	The following conditions can result in this error: • The operation state is not valid: it must be active and fully set up, and this call must conform to the algorithm's requirements for ordering of input and output steps. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_INVALID_SIGNATURE`	The verification fails for a `PSA_PAKE_STEP_ZK_PROOF` or `PSA_PAKE_STEP_CONFIRM` input step.
`PSA_ERROR_INVALID_ARGUMENT`	The following conditions can result in this error: • `step` is not compatible with the operation's algorithm. • The input is not valid for the operation's algorithm, cipher suite or `step`.
`PSA_ERROR_NOT_SUPPORTED`	The following conditions can result in this error: • `step` is not supported with the operation's algorithm. • The input is not supported for the operation's algorithm, cipher suite or `step`.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

Depending on the algorithm being executed, you might need to call this function several times or you might not need to call this at all.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information.

`PSA_PAKE_INPUT_SIZE()` or `PSA_PAKE_INPUT_MAX_SIZE` can be used to allocate buffers of sufficient size to transfer inputs that are received from the peer into the operation.

If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

`psa_pake_get_shared_key` (function)

Extract the shared secret from the PAKE as a key.

Added in version 1.2.

```
psa_status_t psa_pake_get_shared_key(psa_pake_operation_t * operation,
                                     const psa_key_attributes_t * attributes,
                                     psa_key_id_t * key);
```

Parameters

`operation`
Active PAKE operation.

`attributes`
The attributes for the new key.

The following attributes are required for all keys:

- The key type. All PAKE algorithms can output a key of type `PSA_KEY_TYPE_DERIVE` or `PSA_KEY_TYPE_HMAC`. PAKE algorithms that produce a pseudorandom shared secret, can also output block-cipher key types, for example `PSA_KEY_TYPE_AES`. Refer to the documentation of individual PAKE algorithms for more information.

The following attributes must be set for keys used in cryptographic operations:

- The key permitted-algorithm policy, see *Permitted algorithms* on page 101.
- The key usage flags, see *Key usage flags* on page 102.

The following attributes must be set for keys that do not use the default `PSA_KEY_LIFETIME_VOLATILE` lifetime:

- The key lifetime, see *Key lifetimes* on page 90.
- The key identifier is required for a key with a persistent lifetime, see *Key identifiers* on page 98.

The following attributes are optional:

- If the key size is nonzero, it must be equal to the size of the PAKE shared secret.

> **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_BAD_STATE`	The following conditions can result in this error: • The state of PAKE operation `operation` is not valid: it must be ready to return the shared secret. For an unconfirmed key, this will be when the key-exchange output and input steps are complete, but prior to any key-confirmation output and input steps. For a confirmed key, this will be when all key-exchange and key-confirmation output and input steps are complete. • The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_NOT_PERMITTED`	The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.
`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 following conditions can result in this error: • The key type is not valid for output from this operation's algorithm. • The key size is nonzero. • The key lifetime is invalid. • The key identifier is not valid for the key lifetime. • The key usage flags include invalid values. • The key's permitted-usage algorithm is invalid. • The key attributes, as a whole, are invalid.
`PSA_ERROR_NOT_SUPPORTED`	The key attributes, as a whole, are not supported for creation from a PAKE secret, either by the implementation in general or in the specified storage location.
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	
`PSA_ERROR_STORAGE_FAILURE`	
`PSA_ERROR_DATA_CORRUPT`	
`PSA_ERROR_DATA_INVALID`	

Description

The shared secret is retrieved as a key. Its location, policy, and type are taken from `attributes`.

The size of the returned key is always the bit-size of the PAKE shared secret, rounded up to a whole number of bytes. The size of the shared secret is dependent on the PAKE algorithm and cipher suite.

This is the final call in a PAKE operation, which retrieves the shared secret as a key. It is recommended that this key is used as an input to a key-derivation operation to produce additional cryptographic keys. For

some PAKE algorithms, the shared secret is also suitable for use as a key in cryptographic operations such as encryption. Refer to the documentation of individual PAKE algorithms for more information.

Depending on the key confirmation requested in the cipher suite, `psa_pake_get_shared_key()` must be called either before or after the key-confirmation output and input steps for the PAKE algorithm. The key confirmation affects the guarantees that can be made about the shared key:

Unconfirmed key

If the cipher suite used to set up the operation requested an unconfirmed key, the application must call `psa_pake_get_shared_key()` after the key-exchange output and input steps are completed. The PAKE algorithm provides a cryptographic guarantee that only a peer who used the same password, and identity inputs, is able to compute the same key. However, there is no guarantee that the peer is the participant it claims to be, and was able to compute the same key.

Since the peer is not authenticated, no action should be taken that assumes that the peer is who it claims to be. For example, do not access restricted resources on the peer's behalf until an explicit authentication has succeeded.

> **Note:**
> Some PAKE algorithms do not enable the output of the shared secret until it has been confirmed.

Confirmed key

If the cipher suite used to set up the operation requested a confirmed key, the application must call `psa_pake_get_shared_key()` after the key-exchange and key-confirmation output and input steps are completed.

Following key confirmation, the PAKE algorithm provides a cryptographic guarantee that the peer used the same password and identity inputs, and has computed the identical shared secret key.

> **Note:**
> Some PAKE algorithms do not include any key-confirmation steps.

The exact sequence of calls to perform a password-authenticated key exchange depends on the algorithm in use. Refer to the documentation of individual PAKE algorithms for more information.

When this function returns successfully, `operation` becomes inactive. If this function returns an error status, the operation enters an error state and must be aborted by calling `psa_pake_abort()`.

`psa_pake_abort` (function)

Abort a PAKE operation.

Added in version 1.1.

```
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
```

Parameters

`operation`	Initialized PAKE operation.

Returns: `psa_status_t`

`PSA_SUCCESS`	Success. The operation object can now be discarded or reused.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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_pake_setup()` again.

This function can be called any time after the operation object has been initialized as described in `psa_pake_operation_t`.

In particular, calling `psa_pake_abort()` after the operation has been terminated by a call to `psa_pake_abort()` or `psa_pake_get_shared_key()` is safe and has no effect.

10.13.7 PAKE support macros

`PSA_PAKE_OUTPUT_SIZE` (macro)

Sufficient output buffer size for `psa_pake_output()`, in bytes.

Added in version 1.1.

```
#define PSA_PAKE_OUTPUT_SIZE(alg, primitive, output_step) \
    /* implementation-defined value */
```

Parameters

`alg`	A PAKE algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_PAKE(alg)` is true.
`primitive`	A primitive of type `psa_pake_primitive_t` that is compatible with algorithm `alg`.
`output_step`	A value of type `psa_pake_step_t` that is valid for the algorithm `alg`.

Returns

A sufficient output buffer size for the specified PAKE algorithm, primitive, and output step. An implementation can return either `0` or a correct size for a PAKE algorithm, primitive, and output step that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

If the size of the output buffer is at least this large, it is guaranteed that `psa_pake_output()` 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_PAKE_OUTPUT_MAX_SIZE`

PSA_PAKE_OUTPUT_MAX_SIZE (macro)

Sufficient output buffer size for psa_pake_output() for any of the supported PAKE algorithms, primitives and output steps.

Added in version 1.1.

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

If the size of the output buffer is at least this large, it is guaranteed that psa_pake_output() will not fail due to an insufficient buffer size.

See also PSA_PAKE_OUTPUT_SIZE().

PSA_PAKE_INPUT_SIZE (macro)

Sufficient buffer size for inputs to psa_pake_input().

Added in version 1.1.

```
#define PSA_PAKE_INPUT_SIZE(alg, primitive, input_step) \
    /* implementation-defined value */
```

Parameters

alg	A PAKE algorithm: a value of type psa_algorithm_t such that PSA_ALG_IS_PAKE(alg) is true.
primitive	A primitive of type psa_pake_primitive_t that is compatible with algorithm alg.
input_step	A value of type psa_pake_step_t that is valid for the algorithm alg.

Returns

A sufficient buffer size for the specified PAKE algorithm, primitive, and input step. An implementation can return either 0 or a correct size for a PAKE algorithm, primitive, and output step that it recognizes, but does not support. If the parameters are not valid, the return value is unspecified.

Description

The value returned by this macro is guaranteed to be large enough for any valid input to psa_pake_input() in an operation with the specified parameters.

This macro can be useful when transferring inputs from the peer into the PAKE operation.

See also PSA_PAKE_INPUT_MAX_SIZE

PSA_PAKE_INPUT_MAX_SIZE (macro)

Sufficient buffer size for inputs to psa_pake_input() for any of the supported PAKE algorithms, primitives and input steps.

Added in version 1.1.

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

This macro can be useful when transferring inputs from the peer into the PAKE operation.

See also PSA_PAKE_INPUT_SIZE().

10.13.8 The J-PAKE protocol

J-PAKE is the password-authenticated key exchange by juggling protocol, defined by *J-PAKE: Password-Authenticated Key Exchange by Juggling* [RFC8236]. This protocol uses the Schnorr Non-Interactive Zero-Knowledge Proof (NIZKP), as defined by *Schnorr Non-interactive Zero-Knowledge Proof* [RFC8235].

J-PAKE is a balanced PAKE, without key confirmation.

J-PAKE cipher suites

When setting up a PAKE cipher suite to use the J-PAKE protocol:

- Use the `PSA_ALG_JPAKE()` algorithm, parameterized by the required hash algorithm.
- Use a PAKE primitive for the required elliptic curve, or finite field group.
- J-PAKE does not confirm the shared secret key that results from the key exchange.

For example, the following code creates a cipher suite to select J-PAKE using P-256 with the SHA-256 hash function:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_JPAKE(PSA_ALG_SHA_256));
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,
                                             PSA_ECC_FAMILY_SECP_R1, 256));
psa_pake_cs_set_key_confirmation(&cipher_suite, PSA_PAKE_UNCONFIRMED_KEY);
```

More information on selecting a specific elliptic curve or finite field Diffie-Hellman group is provided with the `PSA_PAKE_PRIMITIVE_TYPE_ECC` and `PSA_PAKE_PRIMITIVE_TYPE_DH` constants.

J-PAKE password processing

The PAKE operation for J-PAKE expects a key of type type `PSA_KEY_TYPE_PASSWORD` or PSA_KEY_TYPE_PASSWORD_HASH`. The same key value must be provided to the PAKE operation in both participants.

The key can be the password text itself, in an agreed character encoding, or some value derived from the password, as required by a higher level protocol. For low-entropy passwords, it is recommended that a key-stretching derivation algorithm, such as PBKDF2, is used, and the resulting password hash is used as the key input to the PAKE operation.

J-PAKE operation

The J-PAKE operation follows the protocol shown in Figure 3 on page 367.

Setup

J-PAKE does not assign roles to the participants, so it is not necessary to call `psa_pake_set_role()`.

J-PAKE requires both an application and a peer identity. If the peer identity provided to `psa_pake_set_peer()` does not match the data received from the peer, then the call to `psa_pake_input()` for the `PSA_PAKE_STEP_ZK_PROOF` step will fail with `PSA_ERROR_INVALID_SIGNATURE`.

J-PAKE does not use a context. A call to `psa_pake_set_context()` for a J-PAKE operation will fail with `PSA_ERROR_BAD_STATE`.

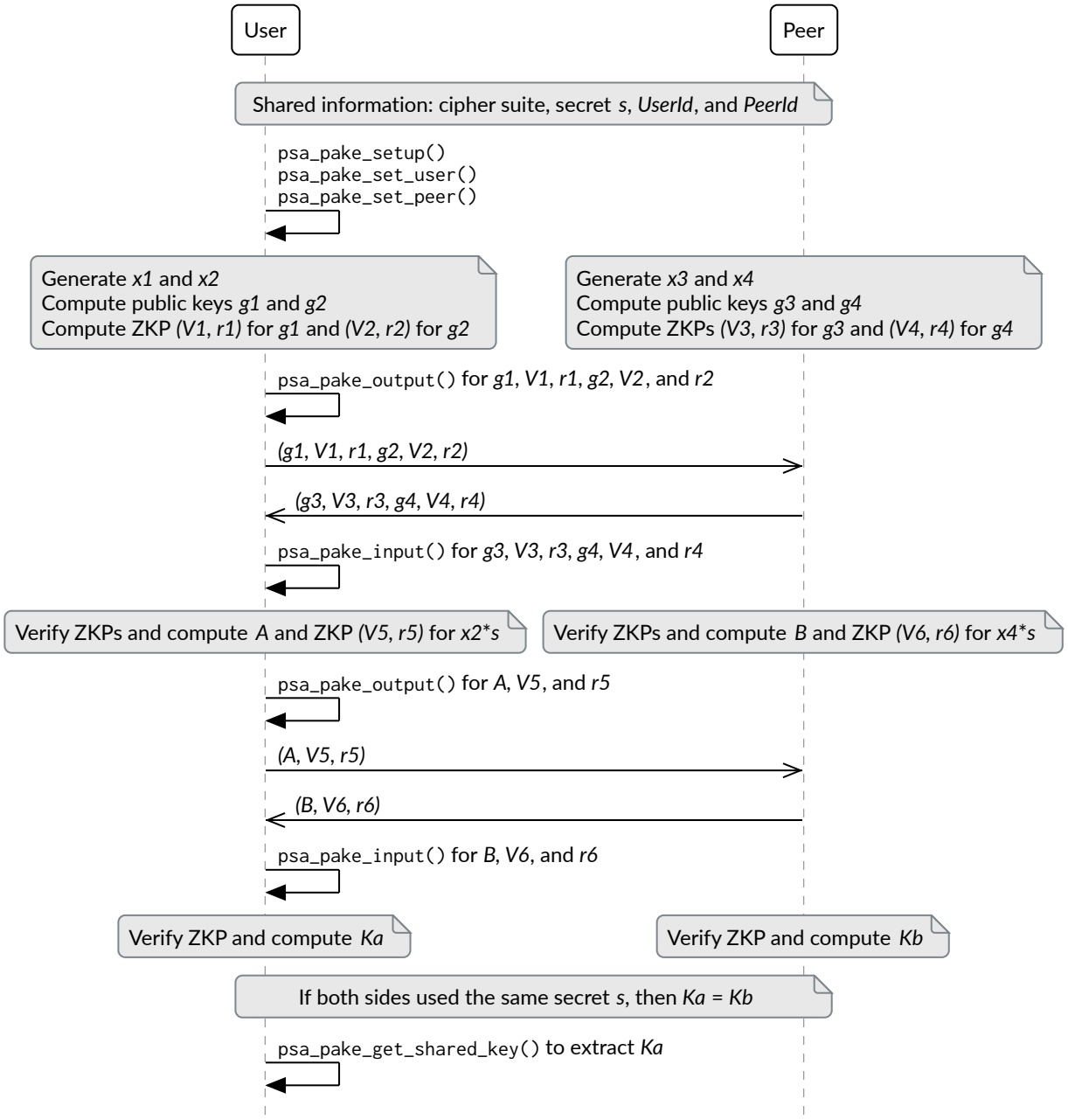

Figure 3 The J-PAKE protocol

The variable names $x1$, $g1$, and so on, are taken from the finite field implementation of J-PAKE in [RFC8236] §2. Details of the computation for the key shares and zero-knowledge proofs are in [RFC8236] and [RFC8235].

The following steps demonstrate the application code for 'User' in Figure 3. The code flow for the 'Peer' is the same as for 'User', as J-PAKE is a balanced PAKE.

1. To prepare a J-PAKE operation, initialize and set up a `psa_pake_operation_t` object by calling the following functions:

```
psa_pake_operation_t jpake = PSA_PAKE_OPERATION_INIT;
```

(continues on next page)

(continued from previous page)

```
psa_pake_setup(&jpake, pake_key, &cipher_suite);
psa_pake_set_user(&jpake, ...);
psa_pake_set_peer(&jpake, ...);
```

See *J-PAKE cipher suites* on page 366 and *J-PAKE password processing* on page 366 for details on the requirements for the cipher suite and key.

The key material is used as an array of bytes, which is converted to an integer as described in *SEC 1: Elliptic Curve Cryptography* [SEC1] §2.3.8, before reducing it modulo q. Here, q is the order of the group defined by the cipher-suite primitive. `psa_pake_setup()` will return an error if the result of the conversion and reduction is `0`.

Key exchange

After setup, the key exchange flow for J-PAKE is as follows:

2. Round one.

   The application can either extract the round one output values first, and then provide the round one inputs that are received from the Peer; or provide the peer inputs first, and then extract the outputs.

   To get the first round data that needs to be sent to the peer, make the following calls to `psa_pake_output()`, in the order shown:

```
// Get g1
psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get V1, the ZKP public key for x1
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r1, the ZKP proof for x1
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
// Get g2
psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get V2, the ZKP public key for x2
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r2, the ZKP proof for x2
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

   To provide the first round data received from the peer to the operation, make the following calls to `psa_pake_input()`, in the order shown:

```
// Set g3
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V3, the ZKP public key for x3
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r3, the ZKP proof for x3
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
// Set g4
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V4, the ZKP public key for x4
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r4, the ZKP proof for x4
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

3. Round two.

   The application can either extract the round two output values first, and then provide the round two inputs that are received from the Peer; or provide the peer inputs first, and then extract the outputs.

   To get the second round data that needs to be sent to the peer, make the following calls to `psa_pake_output()`, in the order shown:

```
// Get A
psa_pake_output(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get V5, the ZKP public key for x2*s
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Get r5, the ZKP proof for x2*s
psa_pake_output(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

   To provide the second round data received from the peer to the operation, make the following calls to `psa_pake_input()`, in the order shown:

```
// Set B
psa_pake_input(&jpake, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set V6, the ZKP public key for x4*s
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PUBLIC, ...);
// Set r6, the ZKP proof for x4*s
psa_pake_input(&jpake, PSA_PAKE_STEP_ZK_PROOF, ...);
```

Extract shared secret

4. To use the shared secret, extract it as a key-derivation key. For example, to extract a derivation key for HKDF-SHA-256:

```
// Set up the key attributes
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
psa_set_key_type(&att, PSA_KEY_TYPE_DERIVE);
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_set_key_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA_256));

// Get Ka=Kb=K
psa_key_id_t shared_key;
psa_pake_get_shared_key(&jpake, &att, &shared_key);
```

For more information about the format of the values which are passed for each step, see *PAKE step types* on page 349.

If the verification of a Zero-knowledge proof provided by the peer fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_ZK_PROOF` step will return `PSA_ERROR_INVALID_SIGNATURE`.

The shared secret that is produced by J-PAKE is not suitable for use as an encryption key. It must be used as an input to a key-derivation operation to produce additional cryptographic keys.

> **Warning**
>
> At the end of this sequence there is a cryptographic guarantee that only a peer that used the same password is able to compute the same key. But there is no guarantee that the peer is the participant it

claims to be, or that the peer used the same password during the exchange.

At this point, authentication is implicit — material encrypted or authenticated using the computed key can only be decrypted or verified by someone with the same key. The peer is not authenticated at this point, and no action should be taken by the application which assumes that the peer is authenticated, for example, by accessing restricted resources.

To make the authentication explicit, there are various methods to confirm that both parties have the same key. See [RFC8236] §5 for two examples.

10.13.9 J-PAKE algorithms

`PSA_ALG_JPAKE` (macro)

Macro to build the Password-authenticated key exchange by juggling (J-PAKE) algorithm.

Added in version 1.1.

Changed in version 1.2: Parameterize J-PAKE algorithm by hash.

```
#define PSA_ALG_JPAKE(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

A J-PAKE algorithm, parameterized by a specific hash.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is J-PAKE as defined by [RFC8236], instantiated with the following parameters:

- The primitive group can be either an elliptic curve or defined over a finite field.
- The Schnorr NIZKP, using the same group as the J-PAKE algorithm.
- The cryptographic hash function, `hash_alg`.

J-PAKE does not confirm the shared secret key that results from the key exchange.

The shared secret that is produced by J-PAKE is not suitable for use as an encryption key. It must be used as an input to a key-derivation operation to produce additional cryptographic keys.

See *The J-PAKE protocol* on page 366 for the J-PAKE protocol flow and how to implement it with the Crypto API.

Compatible key types

`PSA_KEY_TYPE_PASSWORD`

`PSA_KEY_TYPE_PASSWORD_HASH`

`PSA_ALG_IS_JPAKE` (macro)

Whether the specified algorithm is a J-PAKE algorithm.

Added in version 1.2.

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

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

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

Description

J-PAKE algorithms are constructed using `PSA_ALG_JPAKE(hash_alg)`.

10.13.10 The SPAKE2+ protocol

SPAKE2+ is the augmented password-authenticated key exchange protocol, defined by *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383]. SPAKE2+ includes confirmation of the shared secret key that results from the key exchange.

SPAKE2+ is required by *Matter Specification, Version 1.2* [MATTER], as MATTER_PAKE. [MATTER] uses an earlier draft of the SPAKE2+ protocol, *SPAKE2+, an Augmented PAKE (Draft 02)* [SPAKE2P-2].

Although the operation of the PAKE is similar for both of these variants, they have different key schedules for the derivation of the shared secret.

SPAKE2+ cipher suites

SPAKE2+ is instantiated with the following parameters:

- An elliptic curve group.
- A cryptographic hash function.
- A key-derivation function.
- A keyed MAC function.

Valid combinations of these parameters are defined in the table of cipher suites in [RFC9383] §4.

When setting up a PAKE cipher suite to use the SPAKE2+ protocol defined in [RFC9383]:

- For cipher-suites that use HMAC for key confirmation, use the `PSA_ALG_SPAKE2P_HMAC()` algorithm, parameterized by the required hash algorithm.
- For cipher-suites that use CMAC-AES-128 for key confirmation, use the `PSA_ALG_SPAKE2P_CMAC()` algorithm, parameterized by the required hash algorithm.
- Use a PAKE primitive for the required elliptic curve.

For example, the following code creates a cipher suite to select SPAKE2+ using edwards25519 with the SHA-256 hash function:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_SPAKE2P_HMAC(PSA_ALG_SHA_256));
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,
                                             PSA_ECC_FAMILY_TWISTED_EDWARDS, 255));
```

When setting up a PAKE cipher suite to use the SPAKE2+ protocol used by [MATTER]:

- Use the `PSA_ALG_SPAKE2P_MATTER` algorithm.
- Use the `PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC, PSA_ECC_FAMILY_SECP_R1, 256)` PAKE primitive.

The following code creates a cipher suite to select the [MATTER] variant of SPAKE2+:

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_SPAKE2P_MATTER);
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,
                                             PSA_ECC_FAMILY_SECP_R1, 256));
```

SPAKE2+ registration

The SPAKE2+ protocol has distinct roles for the two participants:

- The *Prover* takes the role of client. It uses the protocol to prove that it knows the secret password, and produce a shared secret.
- The *Verifier* takes the role of server. It uses the protocol to verify the client's proof, and produce a shared secret.

The registration phase of SPAKE2+ provides the initial password processing, described in [RFC9383] §3.2. The result of registration is two pairs of values — $(w0, w1)$ and $(w0, L)$ — that need to be provided during the authentication phase to the Prover and Verifier, respectively. The design of SPAKE2+ ensures that knowledge of $(w0, L)$ does not enable an attacker to determine the password, or to compute $w1$.

In the Crypto API, the registration output values are managed as an asymmetric key pair:

- The Prover values, $(w0, w1)$, are stored in a key of type `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`.
- The Verifier values, $(w0, L)$, are stored in a key of type `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()`, or derived from the matching `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR()`.

The SPAKE2+ key types are parameterized by the same elliptic curve as the SPAKE2+ cipher suite.

The key pair is derived from the initial SPAKE2+ password prior to starting the PAKE operation. It is recommended to use a key-stretching derivation algorithm, for example PBKDF2. This process can take place immediately before the PAKE operation, or derived at some earlier point and stored by the participant. Alternatively, the Verifier can be provisioned with the `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY()` for the protocol, by the Prover, or some other agent. Figure 4 on page 373 illustrates some example SPAKE2+ key-derivation flows.

The resulting SPAKE2+ key pair must be protected at least as well as the password. The public key, exported from the key pair, does not need to be kept confidential. It is recommended that the Verifier stores only the public key, because disclosure of the public key does not enable an attacker to impersonate the Prover.

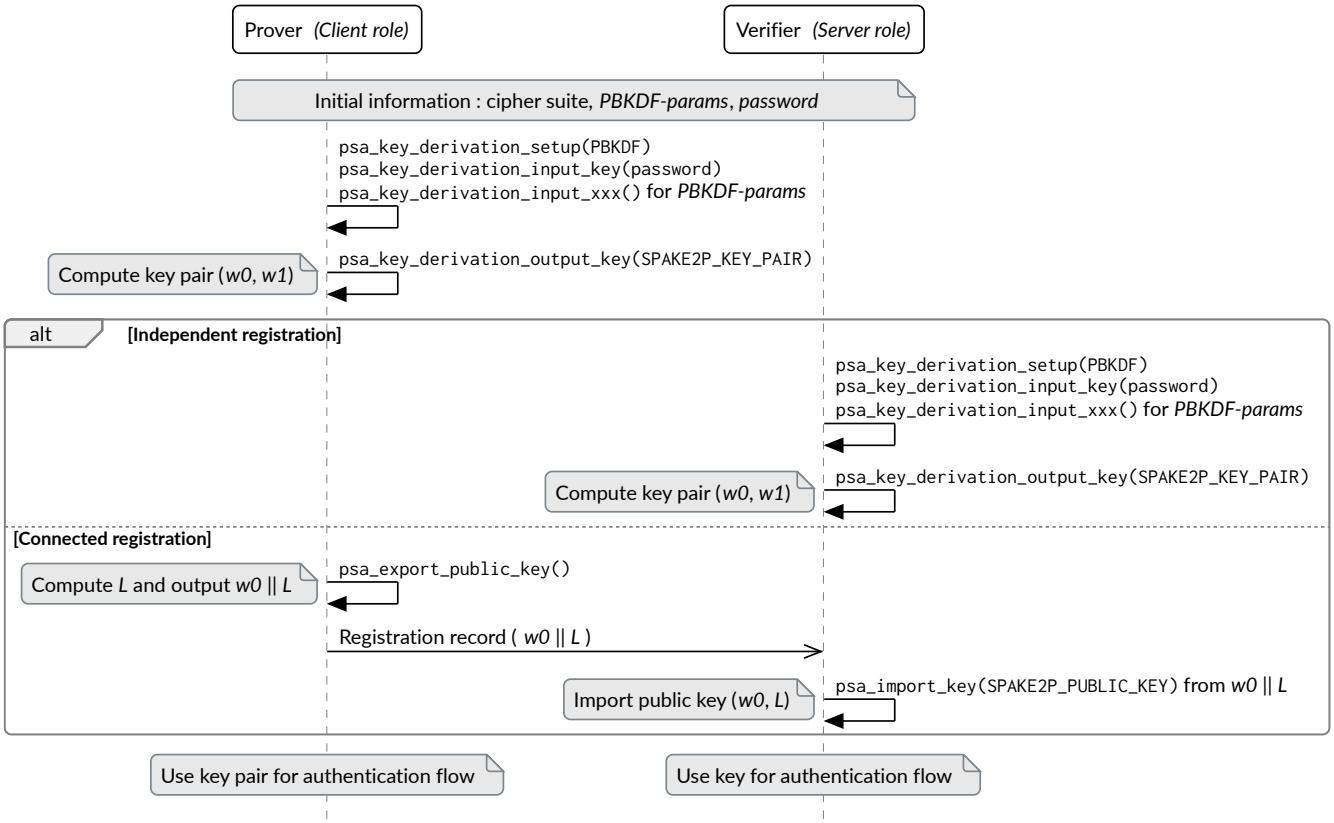

Figure 4 Examples of SPAKE2+ key-derivation procedures

The variable names $w0$, $w1$, and L are taken from the description of SPAKE2+ in [RFC9383].
Details of the computation for the key-derivation values are in [RFC9383] §3.2.

The following steps demonstrate the derivation of a SPAKE2+ key pair using PBKDF2-HMAC-SHA256, for use with a SPAKE2+ cipher suite, `cipher_suite`. See *SPAKE2+ cipher suites* on page 371 for an example of how to construct the cipher suite object.

1. Allocate and initialize a key-derivation object:

```
psa_key_derivation_operation_t pbkdf = PSA_KEY_DERIVATION_OPERATION_INIT;
```

2. Setup the key derivation from the SPAKE2+ password, `password_key`, and parameters `pbkdf2_params`:

```
psa_key_derivation_setup(&pbkdf, PSA_ALG_PBKDF2_HMAC(PSA_ALG_SHA_256));
psa_key_derivation_input_key(&pbkdf, PSA_KEY_DERIVATION_INPUT_PASSWORD, password_key);
psa_key_derivation_input_integer(&pbkdf, PSA_KEY_DERIVATION_INPUT_COST, pbkdf2_params.cost);
psa_key_derivation_input_bytes(&pbkdf, PSA_KEY_DERIVATION_INPUT_SALT,
                                    &pbkdf2_params.salt, pbkdf2_params.salt_len);
```

3. Allocate and initialize a key attributes object:

```
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
```

4. Set the key type, size, and policy from the `cipher_suite` object:

```
const psa_pake_primitive_t primitive = psa_pake_cs_get_primitive(&cipher_suite);

psa_set_key_type(&att,
                 PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(PSA_PAKE_PRIMITIVE_GET_FAMILY(primitive)));
psa_set_key_bits(&att, PSA_PAKE_PRIMITIVE_GET_BITS(primitive));
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_set_key_algorithm(&att, psa_pake_cs_get_algorithm(&cipher_suite));
```

5. Derive the key:

```
psa_key_id_t spake2p_key;
psa_key_derivation_output_key(&att, &pbkdf, &spake2p_key);
psa_key_derivation_abort(&pbkdf);
```

See *SPAKE2+ keys* on page 86 for details of the key types, key-pair derivation, and public-key format.

SPAKE2+ operation

The SPAKE2+ operation follows the protocol shown in Figure 5 on page 375.

Setup

In SPAKE2+, the Prover uses the `PSA_PAKE_ROLE_CLIENT` role, and the Verifier uses the `PSA_PAKE_ROLE_SERVER` role.

The key passed to the Prover must be a SPAKE2+ key pair, which is derived as recommended in *SPAKE2+ registration* on page 372. The key passed to the Verifier can either be a SPAKE2+ key pair, or a SPAKE2+ public key. A SPAKE2+ public key is imported from data that is output by calling `psa_export_public_key()` on a SPAKE2+ key pair.

Both participants in SPAKE2+ have an optional identity. If no identity value is provided, then a zero-length string is used for that identity in the protocol. If the participants do not supply the same identity values to the protocol, the computed secrets will be different, and key confirmation will fail.

Participants in SPAKE2+ can optionally provide a context:

- If `psa_pake_set_context()` is called, then the context and its encoded length are included in the SPAKE2+ transcript computation. This includes the case of a zero-length context.
- If `psa_pake_set_context()` is not called, then the context and its encoded length are omitted entirely from the SPAKE2+ transcript computation. See [RFC9383] §3.3.

If the participants do not supply the same context value to the protocol, the computed secrets will be different, and key confirmation will fail.

The following steps demonstrate the application code for both Prover and Verifier in Figure 5 on page 375.

Prover To prepare a SPAKE2+ operation for the Prover, initialize and set up a `psa_pake_operation_t` object by calling the following functions:

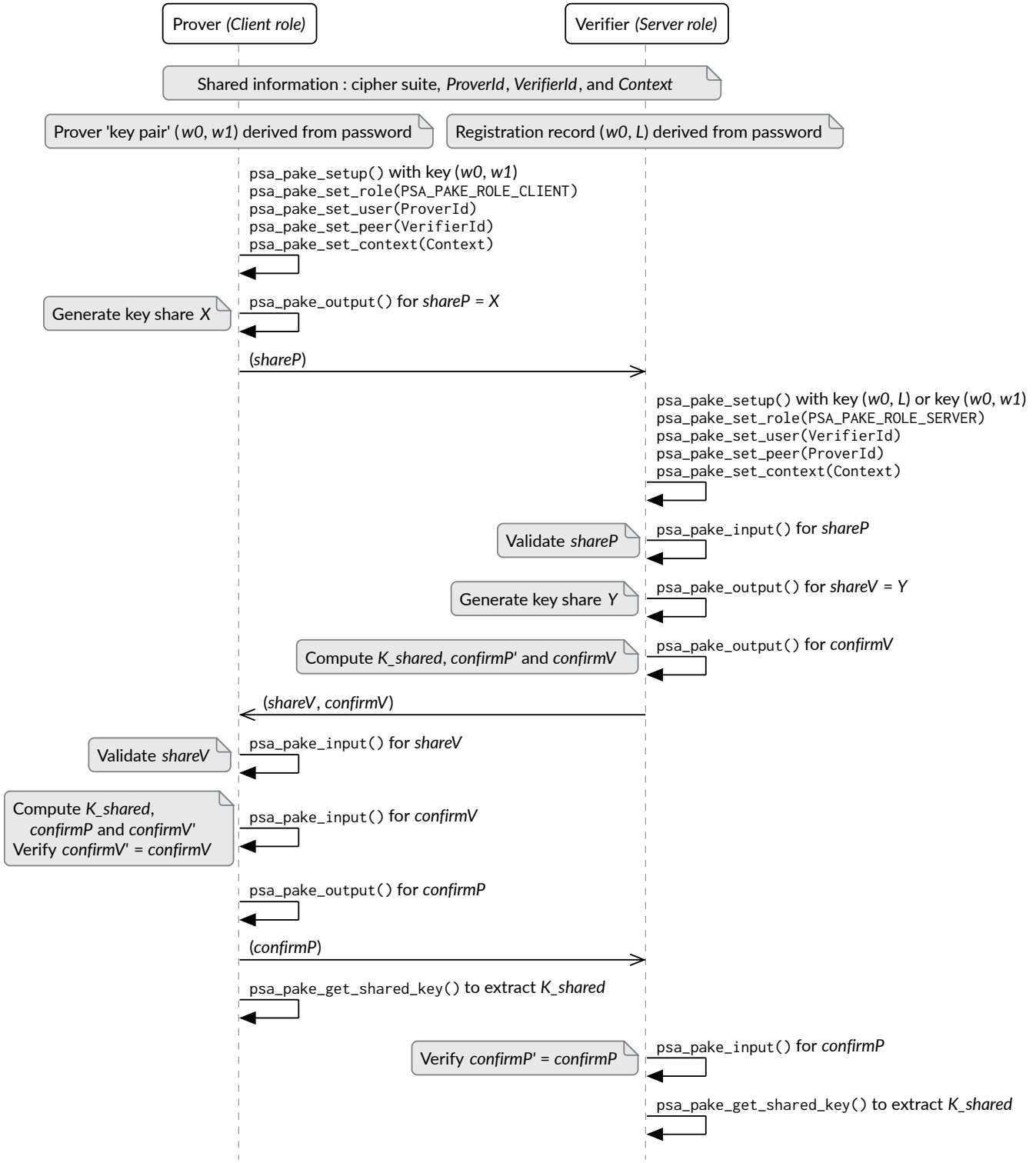

Figure 5 The SPAKE2+ authentication and key confirmation protocol

The variable names $w0$, $w1$, L, and so on, are taken from the description of SPAKE2+ in [RFC9383].
Details of the computation for the key shares is in [RFC9383] §3.3 and confirmation values in [RFC9383] §3.4.

```
psa_pake_operation_t spake2p_p = PSA_PAKE_OPERATION_INIT;

psa_pake_setup(&spake2p_p, pake_key_p, &cipher_suite);
psa_pake_set_role(&spake2p_p, PSA_PAKE_ROLE_CLIENT);
```

The key pake_key_p is a SPAKE2+ key pair, PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(). See *SPAKE2+ cipher suites* on page 371 for details on constructing a suitable cipher suite.

Prover Provide any additional, optional, parameters:

```
psa_pake_set_user(&spake2p_p, ...);       // Prover identity
psa_pake_set_peer(&spake2p_p, ...);       // Verifier identity
psa_pake_set_context(&spake2p_p, ...);    // Optional context
```

Verifier To prepare a SPAKE2+ operation for the Verifier, initialize and set up a psa_pake_operation_t object by calling the following functions:

```
psa_pake_operation_t spake2p_v = PSA_PAKE_OPERATION_INIT;

psa_pake_setup(&spake2p_v, pake_key_v, &cipher_suite);
psa_pake_set_role(&spake2p_v, PSA_PAKE_ROLE_SERVER);
```

The key pake_key_v is a SPAKE2+ key pair, PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(), or public key, PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(). See *SPAKE2+ cipher suites* on page 371 for details on constructing a suitable cipher suite.

Verifier Provide any additional, optional, parameters:

```
psa_pake_set_user(&spake2p_v, ...);       // Verifier identity
psa_pake_set_peer(&spake2p_v, ...);       // Prover identity
psa_pake_set_context(&spake2p_v, ...);    // Optional context
```

Key exchange and confirmation

After setup, the key exchange and confirmation flow for SPAKE2+ is as follows.

> **Note:**
>
> The sequence of calls for the Prover, and the sequence for the Verifier, must be in exactly this order.

Prover To get the key share to send to the Verifier, call:

```
// Get shareP
psa_pake_output(&spake2p_p, PSA_PAKE_STEP_KEY_SHARE, ...);
```

Verifier To provide and validate the key share received from the Prover, call:

```
// Set shareP
psa_pake_input(&spake2p_v, PSA_PAKE_STEP_KEY_SHARE, ...);
```

Verifier To get the Verifier key share and confirmation value to send to the Prover, call:

```
// Get shareV
psa_pake_output(&spake2p_v, PSA_PAKE_STEP_KEY_SHARE, ...);
// Get confirmV
psa_pake_output(&spake2p_v, PSA_PAKE_STEP_CONFIRM, ...);
```

Prover To provide and validate the key share and verify the confirmation value received from the Verifier, call:

```
// Set shareV
psa_pake_input(&spake2p_p, PSA_PAKE_STEP_KEY_SHARE, ...);
// Set confirmV
psa_pake_input(&spake2p_p, PSA_PAKE_STEP_KEY_CONFIRM, ...);
```

Prover To get the Prover key confirmation value to send to the Verifier, call:

```
// Get confirmP
psa_pake_output(&spake2p_p, PSA_PAKE_STEP_CONFIRM, ...);
```

Verifier To verify the confirmation value received from the Prover, call:

```
// Set confirmP
psa_pake_input(&spake2p_v, PSA_PAKE_STEP_CONFIRM, ...);
```

Extract shared secret

Prover To use the shared secret, extract it as a key-derivation key. For example, to extract a derivation key for HKDF-SHA-256:

```
// Set up the key attributes
psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
psa_set_key_type(&att, PSA_KEY_TYPE_DERIVE);
psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
psa_set_key_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA_256));

// Get K_shared
psa_key_id_t shared_key;
psa_pake_get_shared_key(&spake2p_p, &att, &shared_key);
```

Verifier To use the shared secret, extract it as a key-derivation key. The same key attributes can be used as the Prover:

```
// Get K_shared
psa_key_id_t shared_key;
psa_pake_get_shared_key(&spake2p_v, &att, &shared_key);
```

The shared secret that is produced by SPAKE2+ is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key-derivation operation to produce additional cryptographic keys.

For more information about the format of the values which are passed for each step, see *PAKE step types* on page 349.

If the validation of a key share fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_KEY_SHARE` step will return `PSA_ERROR_INVALID_ARGUMENT`. If the verification of a key confirmation value fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_CONFIRM` step will return `PSA_ERROR_INVALID_SIGNATURE`.

10.13.11 SPAKE2+ algorithms

`PSA_ALG_SPAKE2P_HMAC` (macro)

Macro to build the SPAKE2+ algorithm, using HMAC-based key confirmation.

Added in version 1.2.

```
#define PSA_ALG_SPAKE2P_HMAC(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

A SPAKE2+ algorithm, using HMAC for key confirmation, parameterized by a specific hash.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is SPAKE2+, as defined by *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383], for cipher suites that use HMAC for key confirmation. SPAKE2+ cipher suites are specified in [RFC9383] §4. See *SPAKE2+ cipher suites* on page 371.

The shared secret that is produced by SPAKE2+ is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key-derivation operation to produce additional cryptographic keys.

See *The SPAKE2+ protocol* on page 371 for the SPAKE2+ protocol flow and how to implement it with the Crypto API.

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

`PSA_ALG_SPAKE2P_CMAC` (macro)

Macro to build the SPAKE2+ algorithm, using CMAC-based key confirmation.

Added in version 1.2.

```
#define PSA_ALG_SPAKE2P_CMAC(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

A SPAKE2+ algorithm, using CMAC for key confirmation, parameterized by a specific hash.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is SPAKE2+, as defined by *SPAKE2+, an Augmented Password-Authenticated Key Exchange (PAKE) Protocol* [RFC9383], for cipher suites that use CMAC-AES-128 for key confirmation. SPAKE2+ cipher suites are specified in [RFC9383] §4. The cipher suite's hash algorithm is used as input to `PSA_ALG_SPAKE2P_CMAC()`.

The shared secret that is produced by SPAKE2+ is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key-derivation operation to produce additional cryptographic keys.

See *The SPAKE2+ protocol* on page 371 for the SPAKE2+ protocol flow and how to implement it with the Crypto API.

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

`PSA_ALG_SPAKE2P_MATTER` (macro)

The SPAKE2+ algorithm, as used by the Matter v1 specification.

Added in version 1.2.

```
#define PSA_ALG_SPAKE2P_MATTER ((psa_algorithm_t)0x0A000609)
```

This is the PAKE algorithm specified as MATTER_PAKE in *Matter Specification, Version 1.2* [MATTER]. This is based on draft-02 of the SPAKE2+ protocol, *SPAKE2+, an Augmented PAKE (Draft 02)* [SPAKE2P-2]. [MATTER] specifies a single SPAKE2+ cipher suite, P256-SHA256-HKDF-HMAC-SHA256.

The shared secret that is produced by this operation must be processed as directed by the [MATTER] specification.

This algorithm uses the same SPAKE2+ key types, key derivation, protocol flow, and the API usage described in *The SPAKE2+ protocol* on page 371. However, the following aspects are different:

- The key schedule is different. This affects the computation of the shared secret and key confirmation values.
- The protocol inputs and outputs have been renamed between draft-02 and the final RFC, as follows:

RFC 9383	Draft-02
shareP	pA
shareV	pB
confirmP	cA
confirmV	cB
K_shared	Ke

Compatible key types

`PSA_KEY_TYPE_SPAKE2P_KEY_PAIR`

`PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY` (verification only)

`PSA_ALG_IS_SPAKE2P` (macro)

Whether the specified algorithm is a SPAKE2+ algorithm.

Added in version 1.2.

```
#define PSA_ALG_IS_SPAKE2P(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a SPAKE2+ algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

SPAKE2+ algorithms are constructed using `PSA_ALG_SPAKE2P_HMAC(hash_alg)`, `PSA_ALG_SPAKE2P_CMAC(hash_alg)`, or `PSA_ALG_SPAKE2P_MATTER`.

`PSA_ALG_IS_SPAKE2P_HMAC` (macro)

Whether the specified algorithm is a SPAKE2+ algorithm that uses a HMAC-based key confirmation.

Added in version 1.2.

```
#define PSA_ALG_IS_SPAKE2P_HMAC(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a SPAKE2+ algorithm that uses a HMAC-based key confirmation, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

SPAKE2+ algorithms, using HMAC-based key confirmation, are constructed using `PSA_ALG_SPAKE2P_HMAC(hash_alg)`.

`PSA_ALG_IS_SPAKE2P_CMAC` (macro)

Whether the specified algorithm is a SPAKE2+ algorithm that uses a CMAC-based key confirmation.

Added in version 1.2.

```
#define PSA_ALG_IS_SPAKE2P_CMAC(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a SPAKE2+ algorithm that uses a CMAC-based key confirmation, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

SPAKE2+ algorithms, using CMAC-based key confirmation, are constructed using `PSA_ALG_SPAKE2P_CMAC(hash_alg)`.

10.13.12 The WPA3-SAE protocol

WPA3-SAE is a balanced, password-authenticated key exchange protocol, defined by *IEEE 802.11-2024: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications* [IEEE-802.11]. It is used as the authentication and key exchange protocol for WLAN access points and mesh networks. WPA3-SAE includes confirmation of the shared secret key that results from the key exchange.

WPA3-SAE cipher suites

WPA3-SAE is instantiated with the following parameters:

- An elliptic curve group or a finite field cyclic group.
- A cryptographic hash function.

[IEEE-802.11] describes three variants of the WPA3-SAE algorithm. These differ in the method used to generate a password element (PWE) from the password, and in the size of the key confirmation key (SAE-KCK) and pairwise master key (PMK).

Table 17 on page 382 summarizes the properties of the different algorithm variants.

Table 17 WPA3-SAE algorithm variants

Algorithm variant	PWE method	Hash algorithm	SAE-KCK size	PMK size
Looping	Looping	SHA-256	256	256
Hash-to-element	Hash-to-element	SHA-256	256	256
		SHA-384	384	256
		SHA-512	512	256
Group-dependent-hash	Hash-to-element	SHA-256	256	256
		SHA-384	384	384
		SHA-512	512	512

When setting up a PAKE cipher suite to use the WPA3-SAE protocol:

- For the looping variant, use the `PSA_ALG_WPA3_SAE_FIXED(PSA_ALG_SHA_256)` algorithm.
- For the hash-to-element variant, use the `PSA_ALG_WPA3_SAE_FIXED(hash_alg)` algorithm, where `hash_alg` is the required hash algorithm.
- For the group-dependent-hash variant, use the `PSA_ALG_WPA3_SAE_GDH(hash_alg)` algorithm, where `hash_alg` is the required hash algorithm.
- Use a PAKE primitive for the required elliptic curve or finite field group.

Valid elliptic curves and finite field groups for WPA3-SAE are defined in [IEEE-802.11] §12.4.4.1. For the hash-to-element and group-dependent-hash variants, the required hash algorithm is determined from the size of the prime for the cyclic group. See Table 12-1 in [IEEE-802.11] §12.4.2.

If the hash algorithm in the cipher suite is not compatible with the WPA3-SAE algorithm and PAKE primitive, the call to `psa_pake_setup()` will fail with `PSA_ERROR_INVALID_ARGUMENT`.

For example, the following code creates a PAKE cipher suite for WPA3-SAE using hash-to-element over the secp256r1 elliptic curve (IANA group 19):

```
psa_pake_cipher_suite_t cipher_suite = PSA_PAKE_CIPHER_SUITE_INIT;

psa_pake_cs_set_algorithm(&cipher_suite, PSA_ALG_WPA3_SAE_FIXED(PSA_ALG_SHA_256));
psa_pake_cs_set_primitive(&cipher_suite,
                          PSA_PAKE_PRIMITIVE(PSA_PAKE_PRIMITIVE_TYPE_ECC,
                                             PSA_ECC_FAMILY_SECP_R1, 256));
```

WPA3-SAE password processing

WPA3-SAE defines the following two methods for deriving the password element PWE from the password:

Looping method	Repeatedly sample candidate element values using a hash computed from the password, until a valid element is found. This derivation occurs as part of the authentication flow.
Hash-to-element method	Derive a password token element PT from the password, using the hash-to-curve procedure for elliptic curve groups, and a direct method for finite field groups. This derivation can be carried out when the network SSID and password is provisioned to the device, and PT is stored as part of the configuration. During authentication, PWE is derived from PT.

The hash-to-element method is recommended, as it is less vulnerable to timing-based attacks, and reduces the authentication time.

Figure 6 illustrates the password processing required prior to the WPA3-SAE authentication flow.

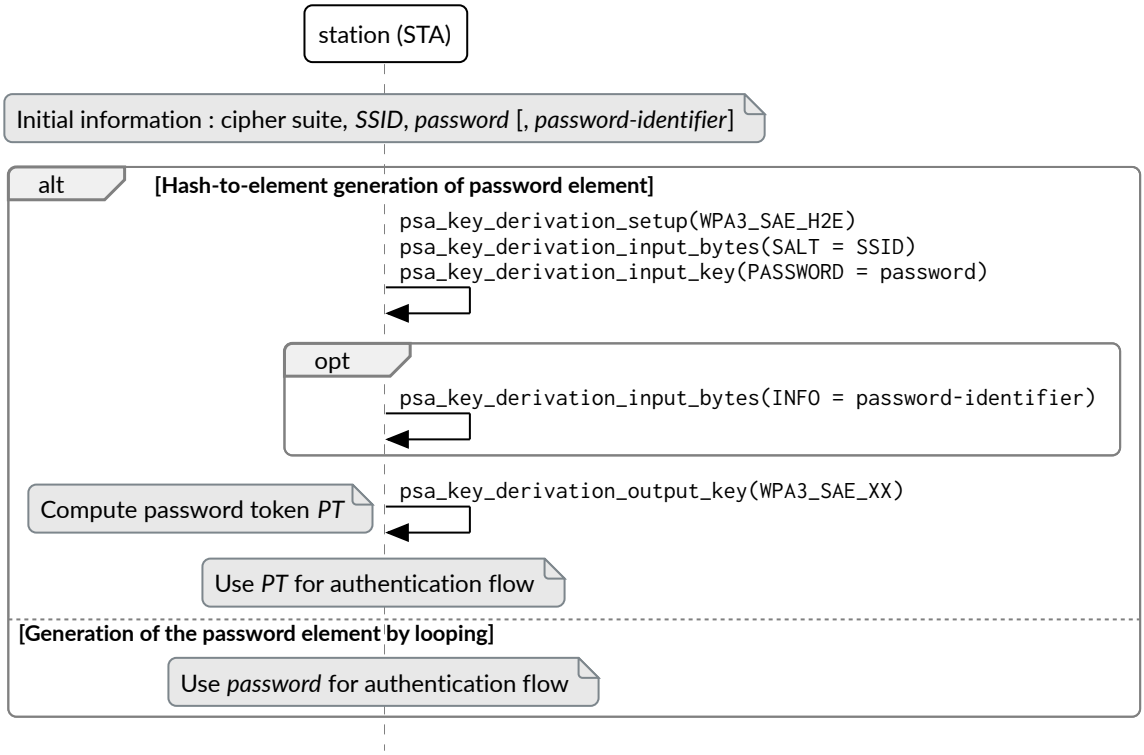

Figure 6 WPA3-SAE password processing

For both methods, the password must be imported as a key of type `PSA_KEY_TYPE_PASSWORD`. The password must be encoded as defined in [IEEE-802.11] §12.4.3.

> **Note:**
>
> [IEEE-802.11] specifies that the same password is used for any configured WPA3-SAE cipher suites, and with any configured PWE-derivation methods. The wildcard key policy `PSA_ALG_WPA3_SAE_ANY` permits a password key to be used for any valid derivation method, and with any valid WPA3-SAE cipher suite.

Looping method

Provide the password key directly to the WPA3-SAE PAKE operation in the call to `psa_pake_setup()`.

Hash-to-element method

To use the hash-to-element method:

1. A WPA3-SAE password token is derived from the WPA3-SAE password, using a key-derivation operation with the `PSA_ALG_WPA3_SAE_H2E()` algorithm. The `PSA_ALG_WPA3_SAE_H2E()` algorithm is parameterized by the hash used in the required WPA3-SAE cipher suite.

The password token is output from the key-derivation operation as a key of type `PSA_KEY_TYPE_WPA3_SAE_ECC()` or `PSA_KEY_TYPE_WPA3_SAE_DH()`. The key type is parameterized by the elliptic curve or finite field Diffie-Hellman group used in the required WPA3-SAE cipher suite.

The password token key must be protected at least as well as the password.

2. Pass the password token key to the WPA3-SAE PAKE operation in the call to `psa_pake_setup()`.

Note:

The wildcard key policy `PSA_ALG_WPA3_SAE_ANY` permits a password token key to be used with both the `PSA_ALG_WPA3_SAE_FIXED()` and `PSA_ALG_WPA3_SAE_GDH()` PAKE algorithms.

The following steps demonstrate the derivation of a password token for use with the group-dependent-hash variant of WPA3-SAE. The selected cipher suite in the example is IANA Group 20: ECC using secp384r1, hash function SHA-384.

1. Allocate and initialize a key-derivation object:

```
psa_key_derivation_operation_t h2e_kdf = PSA_KEY_DERIVATION_OPERATION_INIT;
```

2. Setup the key derivation from the WPA3-SAE password, `password_key`, with network SSID `ssid`:

```
psa_key_derivation_setup(&h2e_kdf, PSA_ALG_WPA3_SAE_H2E(PSA_ALG_SHA_384));
psa_key_derivation_input_bytes(&h2e_kdf, PSA_KEY_DERIVATION_INPUT_SALT, ssid, ssid_len);
psa_key_derivation_input_key(&h2e_kdf, PSA_KEY_DERIVATION_INPUT_PASSWORD, password_key);
```

3. Allocate and initialize a key attributes object:

```
psa_key_attributes_t pt_att = PSA_KEY_ATTRIBUTES_INIT;
```

4. Set the key type, size, and policy:

```
psa_set_key_type(&pt_att,
                 PSA_KEY_TYPE_WPA3_SAE_ECC(PSA_ECC_FAMILY_SECP_R1));
psa_set_key_bits(&pt_att, 384);
psa_set_key_usage_flags(&pt_att, PSA_KEY_USAGE_DERIVE);
psa_set_key_algorithm(&pt_att, PSA_ALG_WPA3_SAE_GDH(PSA_ALG_SHA_384));
```

5. Derive the password token key:

```
psa_key_id_t pt_key;
psa_key_derivation_output_key(&pt_att, &h2e_kdf, &pt_key);
psa_key_derivation_abort(&h2e_kdf);
```

See *WPA3-SAE password tokens* on page 72 for details of the key types and key derivation.

WPA3-SAE operation

The WPA3-SAE authentication operation follows the protocol shown in Figure 7 on page 385.

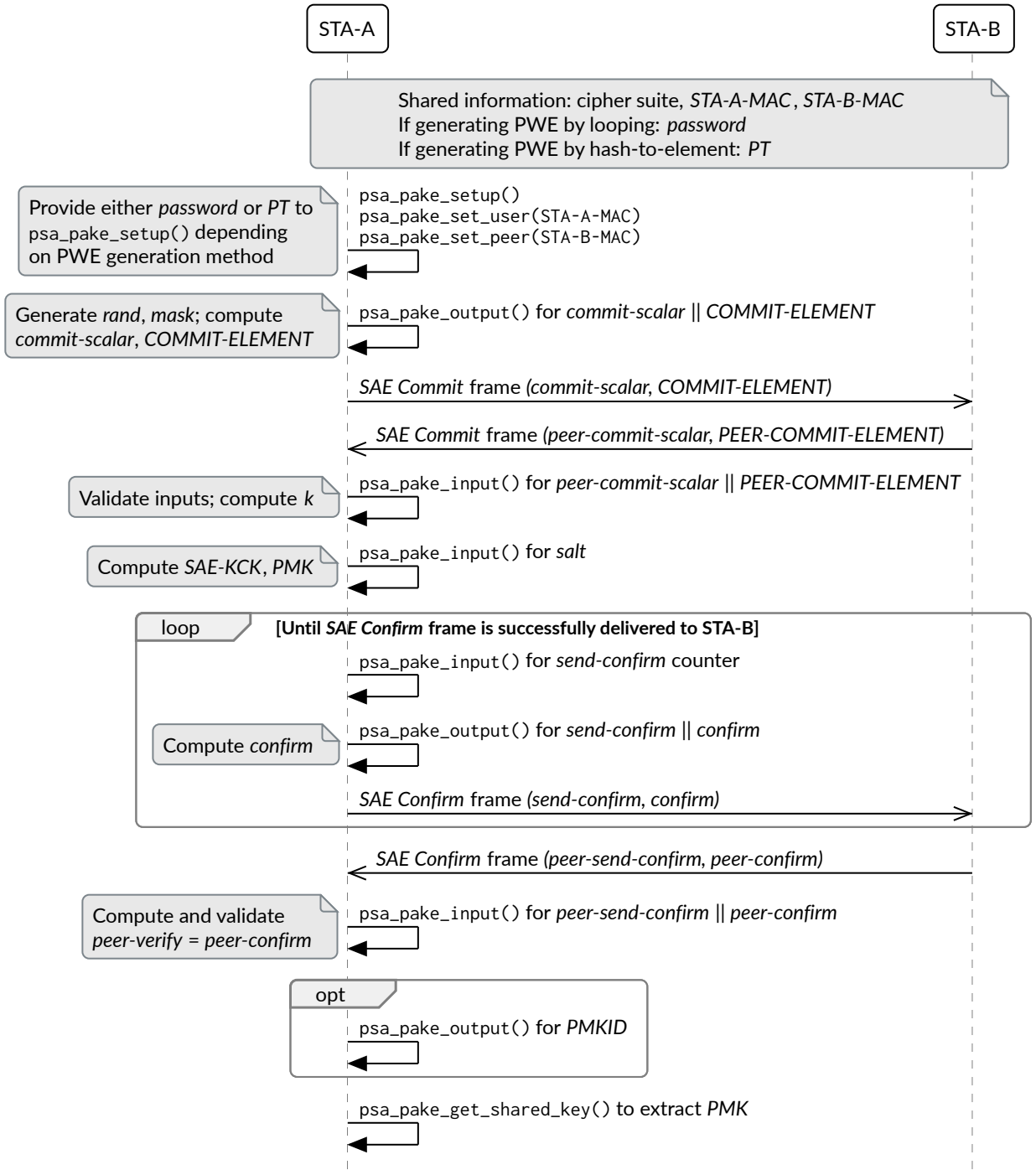

Figure 7 The WPA3-SAE authentication and key confirmation protocol

The variable names *commit-scalar*, *COMMIT-ELEMENT*, *peer-commit-scalar*, and so on, are taken from the description of WPA3-SAE in [IEEE-802.11] §12.4.5.

Setup

The type of keys used to set up a PAKE multi-part operation for WPA3-SAE depends on the variant of WPA3-SAE that is required:

- For the *Looping* variant, use a `PSA_KEY_TYPE_PASSWORD` key containing the secret password.
- For the *Hash-to-element* and *Group-dependent-hash* variants, use a `PSA_KEY_TYPE_WPA3_SAE_ECC` or `PSA_KEY_TYPE_WPA3_SAE_DH` key that is derived from the secret password, as described in *WPA3-SAE password processing* on page 382.

WPA-SAE does not assign roles to the participants, so it is not necessary to call `psa_pake_set_role()`.

WPA-SAE requires the MAC addresses of both participants, which are provided to the PAKE multi-part operation as the user and peer identities.

WPA-SAE does not use a context. A call to `psa_pake_set_context()` for a WPA-SAE operation will fail with `PSA_ERROR_BAD_STATE`.

The following steps demonstrate the application code for STA-A in Figure 7 on page 385. The flow for STA-B is the same as for STA-A, as WPA3-SAE is a balanced PAKE.

1. To prepare a WPA3-SAE operation, initialize and set up a `psa_pake_operation_t` object by calling the following functions:

   ```
   psa_pake_operation_t wpa3_sae = PSA_PAKE_OPERATION_INIT;

   psa_pake_setup(&wpa3_sae, pt_key, &cipher_suite);
   psa_pake_set_user(&wpa3_sae, &sta_a_mac, mac_length);
   psa_pake_set_peer(&wpa3_sae, &sta_b_mac, mac_length);
   ```

   See *WPA3-SAE cipher suites* on page 381 and *WPA3-SAE password processing* on page 382 for details on the requirements for the cipher suite and key.

Commit

2. Exchange commitment values to establish shared secret and confirmation keys.

   The application can either extract the commitment values first, and then provide the commitment values that are received from the peer; or provide the peer inputs first, and then extract the outputs.

   To get the commitment values to send to STA-B, call:

   ```
   // Get commit-scalar || COMMIT-ELEMENT
   psa_pake_output(&wpa3_sae, PSA_PAKE_STEP_COMMIT, ...);
   ```

   To provide and validate the commitment values from STA-B, call:

   ```
   // Set peer-commit-scalar || PEER-COMMIT-ELEMENT
   psa_pake_input(&wpa3_sae, PSA_PAKE_STEP_COMMIT, ...);
   ```

3. Provide the salt used for shared secret derivation, as described in [IEEE-802.11] §12.4.5.4. For Hash-to-element and Group-dependent-hash variants, this is the list of rejected groups.

   ```
   // Set salt
   psa_pake_input(&wpa3_sae, PSA_PAKE_STEP_SALT, ...);
   ```

Confirm

4. Exchange and verify confirmation values.

   WPA3-SAE can make multiple attempts to confirm key establishment, to mitigate frame losses that can occur. To prevent replay of confirmation messages, each attempt generates a distinct confirmation value by including a confirmation counter value.

   The application can either extract a confirmation value first, and then provide a confirmation value received from the peer; or provide the peer input first, and then extract the output.

   To get a confirmation value to send to STA-B, the confirmation counter value $send{-}confirm$ must be updated before extracting the combined *send-confirm* || *confirm* value, as follows:

   ```
   // Set send-confirm counter
   psa_pake_input(&wpa3_sae, PSA_PAKE_STEP_SEND_CONFIRM, ...);
   // Get combined send-confirm || confirm value
   psa_pake_output(&wpa3_sae, PSA_PAKE_STEP_CONFIRM, ...);
   ```

   To verify a confirmation value received from the peer, call:

   ```
   // Set combined peer-send-confirm || peer-confirm value
   psa_pake_input(&wpa3_sae, PSA_PAKE_STEP_CONFIRM, ...);
   ```

   > **Note:**
   > The application is permitted to request new confirmation values, or verify additional peer confirmation values, even after a peer confirmation value has been successfully verified.

Extract shared secret

5. Optionally, to extract the identity of the shared secret key, PMKID, call:

   ```
   // Get PMKID
   psa_pake_output(&wpa3_sae, PSA_PAKE_STEP_KEY_ID, ...);
   ```

6. To use the shared secret, extract it as a key-derivation key. For example, to extract a derivation key for HKDF-SHA-256:

   ```
   // Set up the key attributes
   psa_key_attributes_t att = PSA_KEY_ATTRIBUTES_INIT;
   psa_set_key_type(&att, PSA_KEY_TYPE_DERIVE);
   psa_set_key_usage_flags(&att, PSA_KEY_USAGE_DERIVE);
   psa_set_key_algorithm(&att, PSA_ALG_HKDF(PSA_ALG_SHA_256));

   // Get K_shared
   psa_key_id_t shared_key;
   psa_pake_get_shared_key(&spake2p_p, &att, &shared_key);
   ```

The shared secret that is produced by WPA3-SAE is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key-derivation operation to produce additional cryptographic keys.

For more information about the format of the values which are passed for each step, see *PAKE step types* on page 349.

If the validation of a commitment value fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_COMMIT` step will return `PSA_ERROR_INVALID_ARGUMENT`. If the verification of a confirmation value fails, then the corresponding call to `psa_pake_input()` for the `PSA_PAKE_STEP_CONFIRM` step will return `PSA_ERROR_INVALID_SIGNATURE`.

10.13.13 WPA3-SAE algorithms

`PSA_ALG_WPA3_SAE_FIXED` (macro)

Macro to build the WPA3-SAE algorithm, with fixed-sized PMK output key.

Added in version 1.4.

```
#define PSA_ALG_WPA3_SAE_FIXED(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

A WPA3-SAE algorithm, for the Looping or Hash-to-element variants, parameterized by a specific hash.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is WPA3-SAE, as defined by *IEEE 802.11-2024: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications* [IEEE-802.11] §12.4, using the Looping or Hash-to-element password element derivation procedure, with fixed-sized PMK output key.

The hash algorithm specified must match one of the supported WPA3-SAE cipher suites. See *WPA3-SAE cipher suites* on page 381.

The shared secret that is produced by WPA3-SAE is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key-derivation operation to produce additional cryptographic keys.

See *The WPA3-SAE protocol* on page 381 for the WPA3-SAE protocol flow and how to implement it with the Crypto API.

Compatible key types

`PSA_KEY_TYPE_PASSWORD`

`PSA_KEY_TYPE_WPA3_SAE_ECC`

`PSA_KEY_TYPE_WPA3_SAE_DH`

`PSA_ALG_WPA3_SAE_GDH` (macro)

Macro to build the WPA3-SAE algorithm, with group-dependent size of the PMK output key.

Added in version 1.4.

```
#define PSA_ALG_WPA3_SAE_GDH(hash_alg) /* specification-defined value */
```

Parameters

`hash_alg`	A hash algorithm: a value of type `psa_algorithm_t` such that `PSA_ALG_IS_HASH(hash_alg)` is true.

Returns

A WPA3-SAE algorithm, for the group-dependent-hash variant, parameterized by a specific hash.

Unspecified if `hash_alg` is not a supported hash algorithm.

Description

This is WPA3-SAE, as defined by *IEEE 802.11-2024: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications* [IEEE-802.11] §12.4, using the hash-to-element password element derivation procedure, with group-dependent size for the PMK output key.

The hash algorithm specified must match one of the supported WPA3-SAE cipher suites. See *WPA3-SAE cipher suites* on page 381.

The shared secret that is produced by WPA3-SAE is pseudorandom. Although it can be used directly as an encryption key, it is recommended to use the shared secret as an input to a key-derivation operation to produce additional cryptographic keys.

See *The WPA3-SAE protocol* on page 381 for the WPA3-SAE protocol flow and how to implement it with the Crypto API.

Compatible key types

`PSA_KEY_TYPE_WPA3_SAE_ECC`

`PSA_KEY_TYPE_WPA3_SAE_DH`

`PSA_ALG_IS_WPA3_SAE` (macro)

Whether the specified algorithm is a WPA3-SAE algorithm.

Added in version 1.4.

```
#define PSA_ALG_IS_WPA3_SAE(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a WPA3-SAE algorithm, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

WPA3-SAE algorithms are constructed using `PSA_ALG_WPA3_SAE_FIXED(hash_alg)` or `PSA_ALG_WPA3_SAE_GDH(hash_alg)`.

`PSA_ALG_IS_WPA3_SAE_FIXED` (macro)

Whether the specified algorithm is a WPA3-SAE algorithm with a fixed-sized output key.

Added in version 1.4.

```
#define PSA_ALG_IS_WPA3_SAE_FIXED(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a WPA3-SAE algorithm with a fixed-sized output key, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

WPA3-SAE algorithms with a fixed-sized output key, are constructed using `PSA_ALG_WPA3_SAE_FIXED(hash_alg)`.

`PSA_ALG_IS_WPA3_SAE_GDH` (macro)

Whether the specified algorithm is a WPA3-SAE algorithm with a group-dependent size for the output key.

Added in version 1.4.

```
#define PSA_ALG_IS_WPA3_SAE_GDH(alg) /* specification-defined value */
```

Parameters

`alg`	An algorithm identifier: a value of type `psa_algorithm_t`.

Returns

`1` if `alg` is a WPA3-SAE algorithm with a group-dependent size for the output key, `0` otherwise. This macro can return either `0` or `1` if `alg` is not a supported PAKE algorithm identifier.

Description

WPA3-SAE algorithms with a group-dependent size for the output key, are constructed using `PSA_ALG_WPA3_SAE_GDH(hash_alg)`.

`PSA_ALG_WPA3_SAE_ANY` (macro)

A wildcard algorithm for WPA3-SAE password keys and password token keys.

Added in version 1.4.

```
#define PSA_ALG_WPA3_SAE_ANY ((psa_algorithm_t)0x0a0088ff)
```

If a password key (key type `PSA_KEY_TYPE_PASSWORD`) specifies `PSA_ALG_WPA3_SAE_ANY` as its permitted algorithm, then the key can be used for any WPA3-SAE cipher suite with the `PSA_ALG_WPA3_SAE_H2E` key-derivation algorithm, and with the `PSA_ALG_WPA3_SAE_FIXED` PAKE algorithm.

If a WPA3-SAE password token key specifies `PSA_ALG_WPA3_SAE_ANY` as its permitted algorithm, then the key can be used with both the `PSA_ALG_WPA3_SAE_FIXED()` and `PSA_ALG_WPA3_SAE_GDH()` PAKE algorithms.

10.14 Other cryptographic services

10.14.1 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`	Success. `output` contains `output_size` bytes of generated random data.
`PSA_ERROR_BAD_STATE`	The library requires initializing by a call to `psa_crypto_init()`.
`PSA_ERROR_NOT_SUPPORTED`	
`PSA_ERROR_INSUFFICIENT_ENTROPY`	
`PSA_ERROR_INSUFFICIENT_MEMORY`	
`PSA_ERROR_COMMUNICATION_FAILURE`	
`PSA_ERROR_CORRUPTION_DETECTED`	

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 random key, use `psa_generate_key()` or `psa_generate_key_custom()` instead.

Appendix A: Example header file

Each implementation of the 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.

> **Note:**
>
> Not all of the API elements are fully defined. An implementation must provide the full definition.
>
> The header will not compile without these missing definitions, and might require reordering to satisfy C compilation rules.

A.1 psa/crypto.h

```
/* This file is a reference template for implementation of the
 * PSA Certified Crypto API v1.3
 */

#ifndef PSA_CRYPTO_H
#define PSA_CRYPTO_H

#include <stddef.h>
#include <stdint.h>

#include "psa/error.h"

#ifdef __cplusplus
extern "C" {
#endif

#define PSA_CRYPTO_API_VERSION_MAJOR 1
#define PSA_CRYPTO_API_VERSION_MINOR 4
psa_status_t psa_crypto_init(void);
#define PSA_ERROR_INSUFFICIENT_ENTROPY ((psa_status_t)-148)
#define PSA_ERROR_INVALID_PADDING ((psa_status_t)-150)
typedef uint32_t psa_key_id_t;
typedef /* implementation-defined type */ psa_key_attributes_t;
#define PSA_KEY_ATTRIBUTES_INIT /* implementation-defined value */
psa_key_attributes_t psa_key_attributes_init(void);
psa_status_t psa_get_key_attributes(psa_key_id_t key,
                                    psa_key_attributes_t * attributes);
void psa_reset_key_attributes(psa_key_attributes_t * attributes);
typedef uint16_t psa_key_type_t;
#define PSA_KEY_TYPE_NONE ((psa_key_type_t)0x0000)
#define PSA_KEY_TYPE_IS_UNSTRUCTURED(type) /* specification-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_KEY_TYPE_IS_ASYMMETRIC(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_KEY_PAIR(type) /* specification-defined value */
typedef uint8_t psa_ecc_family_t;
#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_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_TWISTED_EDWARDS ((psa_ecc_family_t) 0x42)
typedef uint8_t psa_dh_family_t;
#define PSA_DH_FAMILY_RFC7919 ((psa_dh_family_t) 0x03)
#define PSA_DH_FAMILY_RFC3526 ((psa_dh_family_t) 0x05)
void psa_set_key_type(psa_key_attributes_t * attributes,
                      psa_key_type_t type);
psa_key_type_t psa_get_key_type(const psa_key_attributes_t * attributes);
size_t psa_get_key_bits(const psa_key_attributes_t * attributes);
void psa_set_key_bits(psa_key_attributes_t * attributes,
                      size_t bits);
#define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x1001)
#define PSA_KEY_TYPE_DERIVE ((psa_key_type_t)0x1200)
#define PSA_KEY_TYPE_PASSWORD ((psa_key_type_t)0x1203)
#define PSA_KEY_TYPE_PASSWORD_HASH ((psa_key_type_t)0x1205)
#define PSA_KEY_TYPE_PEPPER ((psa_key_type_t)0x1206)
#define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x1100)
#define PSA_KEY_TYPE_AES ((psa_key_type_t)0x2400)
#define PSA_KEY_TYPE_ARIA ((psa_key_type_t)0x2406)
#define PSA_KEY_TYPE_DES ((psa_key_type_t)0x2301)
#define PSA_KEY_TYPE_CAMELLIA ((psa_key_type_t)0x2403)
#define PSA_KEY_TYPE_SM4 ((psa_key_type_t)0x2405)
#define PSA_KEY_TYPE_ARC4 ((psa_key_type_t)0x2002)
#define PSA_KEY_TYPE_CHACHA20 ((psa_key_type_t)0x2004)
#define PSA_KEY_TYPE_XCHACHA20 ((psa_key_type_t)0x2007)
#define PSA_KEY_TYPE_ASCON ((psa_key_type_t)0x2008)
#define PSA_KEY_TYPE_WPA3_SAE_ECC(curve) /* specification-defined value */
#define PSA_KEY_TYPE_WPA3_SAE_DH(group) /* specification-defined value */
#define PSA_KEY_TYPE_IS_WPA3_SAE_ECC(type) /* specification-defined value */
#define PSA_KEY_TYPE_WPA3_SAE_ECC_GET_FAMILY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_IS_WPA3_SAE_DH(type) /* specification-defined value */
#define PSA_KEY_TYPE_WPA3_SAE_DH_GET_FAMILY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_RSA_KEY_PAIR ((psa_key_type_t)0x7001)
#define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x4001)
```

(continues on next page)

(continued from previous page)

```
#define PSA_KEY_TYPE_IS_RSA(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_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_ECC_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_IS_DH(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_DH_KEY_PAIR(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_DH_PUBLIC_KEY(type) /* specification-defined value */
#define PSA_KEY_TYPE_DH_GET_FAMILY(type) /* specification-defined value */
#define PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(curve) /* specification-defined value */
#define PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(curve) \
    /* specification-defined value */
#define PSA_KEY_TYPE_IS_SPAKE2P(type) /* specification-defined value */
#define PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) /* specification-defined value */
#define PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type) \
    /* specification-defined value */
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) \
    /* specification-defined value */
typedef uint32_t psa_key_lifetime_t;
typedef uint8_t psa_key_persistence_t;
typedef uint32_t psa_key_location_t;
#define PSA_KEY_LIFETIME_VOLATILE ((psa_key_lifetime_t) 0x00000000)
#define PSA_KEY_LIFETIME_PERSISTENT ((psa_key_lifetime_t) 0x00000001)
#define PSA_KEY_PERSISTENCE_VOLATILE ((psa_key_persistence_t) 0x00)
#define PSA_KEY_PERSISTENCE_DEFAULT ((psa_key_persistence_t) 0x01)
#define PSA_KEY_PERSISTENCE_READ_ONLY ((psa_key_persistence_t) 0xff)
#define PSA_KEY_LOCATION_LOCAL_STORAGE ((psa_key_location_t) 0x000000)
#define PSA_KEY_LOCATION_PRIMARY_SECURE_ELEMENT ((psa_key_location_t) 0x000001)
void psa_set_key_lifetime(psa_key_attributes_t * attributes,
                          psa_key_lifetime_t lifetime);
psa_key_lifetime_t psa_get_key_lifetime(const psa_key_attributes_t * attributes);
#define PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) \
    ((psa_key_persistence_t) ((lifetime) & 0x000000ff))
#define PSA_KEY_LIFETIME_GET_LOCATION(lifetime) \
    ((psa_key_location_t) ((lifetime) >> 8))
#define PSA_KEY_LIFETIME_IS_VOLATILE(lifetime) \
    (PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) == PSA_KEY_PERSISTENCE_VOLATILE)
#define PSA_KEY_LIFETIME_FROM_PERSISTENCE_AND_LOCATION(persistence, location) \
    ((location) << 8 | (persistence))
#define PSA_KEY_ID_NULL ((psa_key_id_t)0)
```

(continues on next page)

(continued from previous page)

```
#define PSA_KEY_ID_USER_MIN ((psa_key_id_t)0x00000001)
#define PSA_KEY_ID_USER_MAX ((psa_key_id_t)0x3fffffff)
#define PSA_KEY_ID_VENDOR_MIN ((psa_key_id_t)0x40000000)
#define PSA_KEY_ID_VENDOR_MAX ((psa_key_id_t)0x7fffffff)
void psa_set_key_id(psa_key_attributes_t * attributes,
                    psa_key_id_t id);
psa_key_id_t psa_get_key_id(const psa_key_attributes_t * attributes);
typedef uint32_t psa_algorithm_t;
void psa_set_key_algorithm(psa_key_attributes_t * attributes,
                           psa_algorithm_t alg);
psa_algorithm_t psa_get_key_algorithm(const psa_key_attributes_t * attributes);
typedef uint32_t psa_key_usage_t;
#define PSA_KEY_USAGE_EXPORT ((psa_key_usage_t)0x00000001)
#define PSA_KEY_USAGE_COPY ((psa_key_usage_t)0x00000002)
#define PSA_KEY_USAGE_CACHE ((psa_key_usage_t)0x00000004)
#define PSA_KEY_USAGE_ENCRYPT ((psa_key_usage_t)0x00000100)
#define PSA_KEY_USAGE_DECRYPT ((psa_key_usage_t)0x00000200)
#define PSA_KEY_USAGE_SIGN_MESSAGE ((psa_key_usage_t)0x00000400)
#define PSA_KEY_USAGE_VERIFY_MESSAGE ((psa_key_usage_t)0x00000800)
#define PSA_KEY_USAGE_SIGN_HASH ((psa_key_usage_t)0x00001000)
#define PSA_KEY_USAGE_VERIFY_HASH ((psa_key_usage_t)0x00002000)
#define PSA_KEY_USAGE_DERIVE ((psa_key_usage_t)0x00004000)
#define PSA_KEY_USAGE_VERIFY_DERIVATION ((psa_key_usage_t)0x00008000)
#define PSA_KEY_USAGE_DERIVE_PUBLIC ((psa_key_usage_t)0x00000080)
#define PSA_KEY_USAGE_WRAP ((psa_key_usage_t)0x00010000)
#define PSA_KEY_USAGE_UNWRAP ((psa_key_usage_t)0x00020000)
void psa_set_key_usage_flags(psa_key_attributes_t * attributes,
                             psa_key_usage_t usage_flags);
psa_key_usage_t psa_get_key_usage_flags(const psa_key_attributes_t * attributes);
psa_status_t psa_check_key_usage(psa_key_id_t key,
                                 psa_algorithm_t alg,
                                 psa_key_usage_t usage);
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);
typedef struct psa_custom_key_parameters_t {
    uint32_t flags;
} psa_custom_key_parameters_t;
#define PSA_CUSTOM_KEY_PARAMETERS_INIT { 0 }
psa_status_t psa_generate_key(const psa_key_attributes_t * attributes,
                              psa_key_id_t * key);
psa_status_t psa_generate_key_custom(const psa_key_attributes_t * attributes,
                                     const psa_custom_key_parameters_t * custom,
                                     const uint8_t * custom_data,
                                     size_t custom_data_length,
                                     psa_key_id_t * key);
psa_status_t psa_copy_key(psa_key_id_t source_key,
```

(continues on next page)

(continued from previous page)

```
                          const psa_key_attributes_t * attributes,
                          psa_key_id_t * target_key);
psa_status_t psa_attach_key(const psa_key_attributes_t * attributes,
                            const uint8_t * label,
                            size_t label_length,
                            psa_key_id_t * key);
psa_status_t psa_destroy_key(psa_key_id_t key);
psa_status_t psa_purge_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);
#define PSA_EXPORT_KEY_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
#define PSA_EXPORT_PUBLIC_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_ASYMMETRIC_KEY_MAX_SIZE /* implementation-defined value */
#define PSA_ALG_NONE ((psa_algorithm_t)0)
#define PSA_ALG_IS_HASH(alg) /* specification-defined value */
#define PSA_ALG_IS_XOF(alg) /* specification-defined value */
#define PSA_ALG_IS_MAC(alg) /* specification-defined value */
#define PSA_ALG_IS_CIPHER(alg) /* specification-defined value */
#define PSA_ALG_IS_AEAD(alg) /* specification-defined value */
#define PSA_ALG_IS_KEY_WRAP(alg) /* specification-defined value */
#define PSA_ALG_IS_KEY_DERIVATION(alg) /* specification-defined value */
#define PSA_ALG_IS_SIGN(alg) /* specification-defined value */
#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) /* specification-defined value */
#define PSA_ALG_IS_KEY_AGREEMENT(alg) /* specification-defined value */
#define PSA_ALG_IS_PAKE(alg) /* specification-defined value */
#define PSA_ALG_IS_KEY_ENCAPSULATION(alg) /* specification-defined value */
#define PSA_ALG_IS_WILDCARD(alg) /* specification-defined value */
#define PSA_ALG_GET_HASH(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_RIPEMD160 ((psa_algorithm_t)0x02000004)
#define PSA_ALG_AES_MMO_ZIGBEE ((psa_algorithm_t)0x02000007)
#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)
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_SHA_512_224 ((psa_algorithm_t)0x0200000c)
#define PSA_ALG_SHA_512_256 ((psa_algorithm_t)0x0200000d)
#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_SHAKE256_512 ((psa_algorithm_t)0x02000015)
#define PSA_ALG_SM3 ((psa_algorithm_t)0x02000014)
#define PSA_ALG_ASCON_HASH256 ((psa_algorithm_t)0x02000019)
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_compare(psa_algorithm_t alg,
                              const uint8_t * input,
                              size_t input_length,
                              const uint8_t * hash,
                              size_t hash_length);
typedef /* implementation-defined type */ psa_hash_operation_t;
#define PSA_HASH_OPERATION_INIT /* implementation-defined value */
psa_hash_operation_t psa_hash_operation_init(void);
psa_status_t psa_hash_setup(psa_hash_operation_t * operation,
                            psa_algorithm_t alg);
psa_status_t psa_hash_update(psa_hash_operation_t * operation,
                             const uint8_t * input,
                             size_t input_length);
psa_status_t psa_hash_finish(psa_hash_operation_t * operation,
                             uint8_t * hash,
                             size_t hash_size,
                             size_t * hash_length);
psa_status_t psa_hash_verify(psa_hash_operation_t * operation,
                             const uint8_t * hash,
                             size_t hash_length);
psa_status_t psa_hash_abort(psa_hash_operation_t * operation);
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_resume(psa_hash_operation_t * operation,
                             const uint8_t * hash_state,
                             size_t hash_state_length);
psa_status_t psa_hash_clone(const psa_hash_operation_t * source_operation,
                            psa_hash_operation_t * target_operation);
#define PSA_HASH_LENGTH(alg) /* implementation-defined value */
#define PSA_HASH_MAX_SIZE /* implementation-defined value */
#define PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) /* specification-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH ((size_t)4)
#define PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) \
    /* specification-defined value */
#define PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) \
    /* specification-defined value */
#define PSA_HASH_BLOCK_LENGTH(alg) /* implementation-defined value */
#define PSA_ALG_SHAKE128 ((psa_algorithm_t)0x0D000100)
#define PSA_ALG_SHAKE256 ((psa_algorithm_t)0x0D000200)
#define PSA_ALG_ASCON_XOF128 ((psa_algorithm_t)0x0D000300)
#define PSA_ALG_ASCON_CXOF128 ((psa_algorithm_t)0x0D008300)
typedef /* implementation-defined type */ psa_xof_operation_t;
#define PSA_XOF_OPERATION_INIT /* implementation-defined value */
psa_xof_operation_t psa_xof_operation_init(void);
psa_status_t psa_xof_setup(psa_xof_operation_t * operation,
                           psa_algorithm_t alg);
psa_status_t psa_xof_set_context(psa_xof_operation_t * operation,
                                 const uint8_t * context,
                                 size_t context_length);
psa_status_t psa_xof_update(psa_xof_operation_t * operation,
                            const uint8_t * input,
                            size_t input_length);
psa_status_t psa_xof_output(psa_xof_operation_t * operation,
                            uint8_t * output,
                            size_t output_length);
psa_status_t psa_xof_abort(psa_xof_operation_t * operation);
#define PSA_ALG_XOF_HAS_CONTEXT(alg) /* specification-defined value */
#define PSA_ALG_HMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_CBC_MAC ((psa_algorithm_t)0x03c00100)
#define PSA_ALG_CMAC ((psa_algorithm_t)0x03c00200)
#define PSA_ALG_TRUNCATED_MAC(mac_alg, mac_length) \
    /* specification-defined value */
#define PSA_ALG_FULL_LENGTH_MAC(mac_alg) /* specification-defined value */
#define PSA_ALG_AT_LEAST_THIS_LENGTH_MAC(mac_alg, min_mac_length) \
    /* specification-defined value */
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_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);
```

(continues on next page)

(continued from previous page)

```
typedef /* implementation-defined type */ psa_mac_operation_t;
#define PSA_MAC_OPERATION_INIT /* implementation-defined value */
psa_mac_operation_t psa_mac_operation_init(void);
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_verify_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_sign_finish(psa_mac_operation_t * operation,
                                 uint8_t * mac,
                                 size_t mac_size,
                                 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_abort(psa_mac_operation_t * operation);
#define PSA_ALG_IS_HMAC(alg) /* specification-defined value */
#define PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) /* specification-defined value */
#define PSA_MAC_LENGTH(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_MAC_MAX_SIZE /* implementation-defined value */
#define PSA_ALG_STREAM_CIPHER ((psa_algorithm_t)0x04800100)
#define PSA_ALG_CTR ((psa_algorithm_t)0x04c01000)
#define PSA_ALG_CCM_STAR_NO_TAG ((psa_algorithm_t)0x04c01300)
#define PSA_ALG_CFB ((psa_algorithm_t)0x04c01100)
#define PSA_ALG_OFB ((psa_algorithm_t)0x04c01200)
#define PSA_ALG_XTS ((psa_algorithm_t)0x0440ff00)
#define PSA_ALG_ECB_NO_PADDING ((psa_algorithm_t)0x04404400)
#define PSA_ALG_CBC_NO_PADDING ((psa_algorithm_t)0x04404000)
#define PSA_ALG_CBC_PKCS7 ((psa_algorithm_t)0x04404100)
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_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);
```

(continues on next page)

(continued from previous page)

```
typedef /* implementation-defined type */ psa_cipher_operation_t;
#define PSA_CIPHER_OPERATION_INIT /* implementation-defined value */
psa_cipher_operation_t psa_cipher_operation_init(void);
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_decrypt_setup(psa_cipher_operation_t * operation,
                                      psa_key_id_t key,
                                      psa_algorithm_t alg);
psa_status_t psa_cipher_generate_iv(psa_cipher_operation_t * operation,
                                    uint8_t * iv,
                                    size_t iv_size,
                                    size_t * iv_length);
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,
                               uint8_t * output,
                               size_t output_size,
                               size_t * output_length);
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_abort(psa_cipher_operation_t * operation);
#define PSA_ALG_IS_STREAM_CIPHER(alg) /* specification-defined value */
#define PSA_ALG_CCM_STAR_ANY_TAG ((psa_algorithm_t)0x04c09300)
#define PSA_CIPHER_ENCRYPT_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_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length) \
    /* 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_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_UPDATE_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
#define PSA_CIPHER_FINISH_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
#define PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_BLOCK_CIPHER_BLOCK_LENGTH(type) /* specification-defined value */
#define PSA_BLOCK_CIPHER_BLOCK_MAX_SIZE /* implementation-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_CCM ((psa_algorithm_t)0x05500100)
#define PSA_ALG_GCM ((psa_algorithm_t)0x05500200)
#define PSA_ALG_CHACHA20_POLY1305 ((psa_algorithm_t)0x05100500)
#define PSA_ALG_XCHACHA20_POLY1305 ((psa_algorithm_t)0x05100600)
#define PSA_ALG_ASCON_AEAD128 ((psa_algorithm_t)0x05100700)
#define PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, tag_length) \
    /* specification-defined value */
#define PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG(aead_alg) \
    /* specification-defined value */
#define PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG(aead_alg, min_tag_length) \
    /* specification-defined value */
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_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);
typedef /* implementation-defined type */ psa_aead_operation_t;
#define PSA_AEAD_OPERATION_INIT /* implementation-defined value */
psa_aead_operation_t psa_aead_operation_init(void);
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_decrypt_setup(psa_aead_operation_t * operation,
                                    psa_key_id_t key,
                                    psa_algorithm_t alg);
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_generate_nonce(psa_aead_operation_t * operation,
                                     uint8_t * nonce,
                                     size_t nonce_size,
```

(continues on next page)

(continued from previous page)

```
                                    size_t * nonce_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_ad(psa_aead_operation_t * operation,
                                const uint8_t * input,
                                size_t input_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_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_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_aead_abort(psa_aead_operation_t * operation);
#define PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER(alg) /* specification-defined value */
#define PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length) \
    /* implementation-defined value */
#define PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length) \
    /* implementation-defined value */
#define PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg, ciphertext_length) \
    /* implementation-defined value */
#define PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length) \
    /* 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_UPDATE_OUTPUT_SIZE(key_type, alg, input_length) \
    /* implementation-defined value */
#define PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length) \
    /* implementation-defined value */
#define PSA_AEAD_FINISH_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
#define PSA_AEAD_FINISH_OUTPUT_MAX_SIZE /* 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 */
```

(continues on next page)

(continued from previous page)

```
#define PSA_AEAD_VERIFY_OUTPUT_SIZE(key_type, alg) \
    /* implementation-defined value */
#define PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_ALG_KW ((psa_algorithm_t)0x0B400100)
#define PSA_ALG_KWP ((psa_algorithm_t)0x0BC00200)
psa_status_t psa_unwrap_key(const psa_key_attributes_t * attributes,
                            psa_key_id_t wrapping_key,
                            psa_algorithm_t alg,
                            const uint8_t * data,
                            size_t data_length,
                            psa_key_id_t * key);
psa_status_t psa_wrap_key(psa_key_id_t wrapping_key,
                          psa_algorithm_t alg,
                          psa_key_id_t key,
                          uint8_t * data,
                          size_t data_size,
                          size_t * data_length);
#define PSA_WRAP_KEY_OUTPUT_SIZE(wrap_key_type, alg, key_type, key_bits) \
    /* implementation-defined value */
#define PSA_WRAP_KEY_PAIR_MAX_SIZE /* implementation-defined value */
#define PSA_ALG_HKDF(hash_alg) /* specification-defined value */
#define PSA_ALG_HKDF_EXTRACT(hash_alg) /* specification-defined value */
#define PSA_ALG_HKDF_EXPAND(hash_alg) /* specification-defined value */
#define PSA_ALG_SP800_108_COUNTER_HMAC(hash_alg) \
    /* specification-defined value */
#define PSA_ALG_SP800_108_COUNTER_CMAC ((psa_algorithm_t)0x08000800)
#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_TLS12_ECJPAKE_TO_PMS ((psa_algorithm_t)0x08000609)
#define PSA_ALG_WPA3_SAE_H2E(hash_alg) /* specification-defined value */
#define PSA_ALG_PBKDF2_HMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_PBKDF2_AES_CMAC_PRF_128 ((psa_algorithm_t)0x08800200)
typedef uint16_t psa_key_derivation_step_t;
#define PSA_KEY_DERIVATION_INPUT_SECRET /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_OTHER_SECRET \
    /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_PASSWORD /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_LABEL /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_CONTEXT /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_SALT /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_INFO /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_SEED /* implementation-defined value */
#define PSA_KEY_DERIVATION_INPUT_COST /* implementation-defined value */
typedef /* implementation-defined type */ psa_key_derivation_operation_t;
#define PSA_KEY_DERIVATION_OPERATION_INIT /* implementation-defined value */
psa_key_derivation_operation_t psa_key_derivation_operation_init(void);
psa_status_t psa_key_derivation_setup(psa_key_derivation_operation_t * operation,
                                      psa_algorithm_t alg);
```

(continues on next page)

(continued from previous page)

```
psa_status_t psa_key_derivation_get_capacity(const psa_key_derivation_operation_t * operation,
                                             size_t * capacity);
psa_status_t psa_key_derivation_set_capacity(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_integer(psa_key_derivation_operation_t * operation,
                                              psa_key_derivation_step_t step,
                                              uint64_t value);
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_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_output_key_custom(const psa_key_attributes_t * attributes,
                                                  psa_key_derivation_operation_t * operation,
                                                  const psa_custom_key_parameters_t * custom,
                                                  const uint8_t * custom_data,
                                                  size_t custom_data_length,
                                                  psa_key_id_t * key);
psa_status_t psa_key_derivation_verify_bytes(psa_key_derivation_operation_t * operation,
                                             const uint8_t * expected_output,
                                             size_t output_length);
psa_status_t psa_key_derivation_verify_key(psa_key_derivation_operation_t * operation,
                                           psa_key_id_t expected);
psa_status_t psa_key_derivation_abort(psa_key_derivation_operation_t * operation);
#define PSA_ALG_IS_KEY_DERIVATION_STRETCHING(alg) \
    /* specification-defined value */
#define PSA_ALG_IS_HKDF(alg) /* specification-defined value */
#define PSA_ALG_IS_HKDF_EXTRACT(alg) /* specification-defined value */
#define PSA_ALG_IS_HKDF_EXPAND(alg) /* specification-defined value */
#define PSA_ALG_IS_SP800_108_COUNTER_HMAC(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_PBKDF2_HMAC(alg) /* specification-defined value */
#define PSA_ALG_IS_WPA3_SAE_H2E(alg) /* specification-defined value */
#define PSA_KEY_DERIVATION_UNLIMITED_CAPACITY \
    /* implementation-defined value */
#define PSA_TLS12_PSK_TO_MS_PSK_MAX_SIZE /* implementation-defined value */
#define PSA_TLS12_ECJPAKE_TO_PMS_OUTPUT_SIZE 32
#define PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg) /* specification-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_RSA_PKCS1V15_SIGN_RAW ((psa_algorithm_t) 0x06000200)
#define PSA_ALG_RSA_PSS(hash_alg) /* specification-defined value */
#define PSA_ALG_RSA_PSS_ANY_SALT(hash_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_RSA_PSS_ANY_SALT(alg) /* specification-defined value */
#define PSA_ALG_IS_RSA_PSS_STANDARD_SALT(alg) /* specification-defined value */
#define PSA_ALG_ECDSA(hash_alg) /* specification-defined value */
#define PSA_ALG_ECDSA_ANY ((psa_algorithm_t) 0x06000600)
#define PSA_ALG_DETERMINISTIC_ECDSA(hash_alg) /* specification-defined value */
#define PSA_ALG_IS_ECDSA(alg) /* specification-defined value */
#define PSA_ALG_IS_DETERMINISTIC_ECDSA(alg) /* specification-defined value */
#define PSA_ALG_IS_RANDOMIZED_ECDSA(alg) /* specification-defined value */
#define PSA_ALG_PURE_EDDSA ((psa_algorithm_t) 0x06000800)
#define PSA_ALG_EDDSA_CTX ((psa_algorithm_t) 0x06000A00)
#define PSA_ALG_ED25519PH ((psa_algorithm_t) 0x0600090B)
#define PSA_ALG_ED448PH ((psa_algorithm_t) 0x06000915)
#define PSA_ALG_IS_HASH_EDDSA(alg) /* specification-defined value */
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);
psa_status_t psa_sign_message_with_context(psa_key_id_t key,
                                           psa_algorithm_t alg,
                                           const uint8_t * input,
                                           size_t input_length,
                                           const uint8_t * context,
                                           size_t context_length,
                                           uint8_t * signature,
                                           size_t signature_size,
                                           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);
psa_status_t psa_verify_message_with_context(psa_key_id_t key,
                                             psa_algorithm_t alg,
                                             const uint8_t * input,
                                             size_t input_length,
                                             const uint8_t * context,
                                             size_t context_length,
                                             const uint8_t * signature,
                                             size_t signature_length);
```

(continues on next page)

(continued from previous page)

```
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_hash_with_context(psa_key_id_t key,
                                        psa_algorithm_t alg,
                                        const uint8_t * hash,
                                        size_t hash_length,
                                        const uint8_t * context,
                                        size_t context_length,
                                        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_hash_with_context(psa_key_id_t key,
                                          psa_algorithm_t alg,
                                          const uint8_t * hash,
                                          size_t hash_length,
                                          const uint8_t * context,
                                          size_t context_length,
                                          const uint8_t * signature,
                                          size_t signature_length);
#define PSA_ALG_IS_SIGN_MESSAGE(alg) /* specification-defined value */
#define PSA_ALG_IS_SIGN_HASH(alg) /* specification-defined value */
#define PSA_ALG_IS_HASH_AND_SIGN(alg) /* specification-defined value */
#define PSA_ALG_SIGN_SUPPORTS_CONTEXT(alg) /* implementation-defined value */
#define PSA_ALG_ANY_HASH ((psa_algorithm_t)0x020000ff)
#define PSA_SIGN_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_SIGNATURE_MAX_SIZE /* implementation-defined value */
#define PSA_ALG_RSA_PKCS1V15_CRYPT ((psa_algorithm_t)0x07000200)
#define PSA_ALG_RSA_OAEP(hash_alg) /* specification-defined value */
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,
```

(continues on next page)

(continued from previous page)

```
                                    size_t * output_length);
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);
#define PSA_ALG_IS_RSA_OAEP(alg) /* specification-defined value */
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_ASYMMETRIC_DECRYPT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
#define PSA_ALG_FFDH ((psa_algorithm_t)0x09010000)
#define PSA_ALG_ECDH ((psa_algorithm_t)0x09020000)
#define PSA_ALG_KEY_AGREEMENT(ka_alg, kdf_alg) \
    /* specification-defined value */
psa_status_t psa_key_agreement(psa_key_id_t private_key,
                               const uint8_t * peer_key,
                               size_t peer_key_length,
                               psa_algorithm_t alg,
                               const psa_key_attributes_t * attributes,
                               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);
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);
#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_IS_STANDALONE_KEY_AGREEMENT(alg) \
    /* specification-defined value */
#define PSA_ALG_IS_RAW_KEY_AGREEMENT(alg) \
    PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg)
#define PSA_ALG_IS_FFDH(alg) /* specification-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_IS_ECDH(alg) /* specification-defined value */
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE \
    /* implementation-defined value */
#define PSA_ALG_ECIES_SEC1 ((psa_algorithm_t)0x0c000100)
psa_status_t psa_encapsulate(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const psa_key_attributes_t * attributes,
                             psa_key_id_t * output_key,
                             uint8_t * ciphertext,
                             size_t ciphertext_size,
                             size_t * ciphertext_length);
psa_status_t psa_decapsulate(psa_key_id_t key,
                             psa_algorithm_t alg,
                             const uint8_t * ciphertext,
                             size_t ciphertext_length,
                             const psa_key_attributes_t * attributes,
                             psa_key_id_t * output_key);
#define PSA_ENCAPSULATE_CIPHERTEXT_SIZE(key_type, key_bits, alg) \
    /* implementation-defined value */
#define PSA_ENCAPSULATE_CIPHERTEXT_MAX_SIZE /* implementation-defined value */
typedef uint32_t psa_pake_primitive_t;
typedef uint8_t psa_pake_primitive_type_t;
#define PSA_PAKE_PRIMITIVE_TYPE_ECC ((psa_pake_primitive_type_t)0x01)
#define PSA_PAKE_PRIMITIVE_TYPE_DH ((psa_pake_primitive_type_t)0x02)
typedef uint8_t psa_pake_family_t;
#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_GET_TYPE(pake_primitive) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_GET_FAMILY(pake_primitive) \
    /* specification-defined value */
#define PSA_PAKE_PRIMITIVE_GET_BITS(pake_primitive) \
    /* specification-defined value */
typedef /* implementation-defined type */ psa_pake_cipher_suite_t;
#define PSA_PAKE_CIPHER_SUITE_INIT /* implementation-defined value */
psa_pake_cipher_suite_t psa_pake_cipher_suite_init(void);
psa_algorithm_t psa_pake_cs_get_algorithm(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_algorithm(psa_pake_cipher_suite_t* cipher_suite,
                               psa_algorithm_t alg);
psa_pake_primitive_t psa_pake_cs_get_primitive(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_primitive(psa_pake_cipher_suite_t* cipher_suite,
                               psa_pake_primitive_t primitive);
#define PSA_PAKE_CONFIRMED_KEY 0
#define PSA_PAKE_UNCONFIRMED_KEY 1
uint32_t psa_pake_cs_get_key_confirmation(const psa_pake_cipher_suite_t* cipher_suite);
void psa_pake_cs_set_key_confirmation(psa_pake_cipher_suite_t* cipher_suite,
```

(continues on next page)

(continued from previous page)

```
                                    uint32_t key_confirmation);
typedef uint8_t psa_pake_role_t;
#define PSA_PAKE_ROLE_NONE ((psa_pake_role_t)0x00)
#define PSA_PAKE_ROLE_FIRST ((psa_pake_role_t)0x01)
#define PSA_PAKE_ROLE_SECOND ((psa_pake_role_t)0x02)
#define PSA_PAKE_ROLE_CLIENT ((psa_pake_role_t)0x11)
#define PSA_PAKE_ROLE_SERVER ((psa_pake_role_t)0x12)
typedef uint8_t psa_pake_step_t;
#define PSA_PAKE_STEP_KEY_SHARE ((psa_pake_step_t)0x01)
#define PSA_PAKE_STEP_ZK_PUBLIC ((psa_pake_step_t)0x02)
#define PSA_PAKE_STEP_ZK_PROOF ((psa_pake_step_t)0x03)
#define PSA_PAKE_STEP_CONFIRM ((psa_pake_step_t)0x04)
#define PSA_PAKE_STEP_SALT ((psa_pake_step_t)0x05)
#define PSA_PAKE_STEP_COMMIT ((psa_pake_step_t)0x06)
#define PSA_PAKE_STEP_CONFIRM_COUNT ((psa_pake_step_t)0x07)
#define PSA_PAKE_STEP_KEY_ID ((psa_pake_step_t)0x08)
typedef /* implementation-defined type */ psa_pake_operation_t;
#define PSA_PAKE_OPERATION_INIT /* implementation-defined value */
psa_pake_operation_t psa_pake_operation_init(void);
psa_status_t psa_pake_setup(psa_pake_operation_t * operation,
                            psa_key_id_t password_key,
                            const psa_pake_cipher_suite_t * cipher_suite);
psa_status_t psa_pake_set_role(psa_pake_operation_t * operation,
                               psa_pake_role_t role);
psa_status_t psa_pake_set_user(psa_pake_operation_t * operation,
                               const uint8_t * user_id,
                               size_t user_id_len);
psa_status_t psa_pake_set_peer(psa_pake_operation_t * operation,
                               const uint8_t * peer_id,
                               size_t peer_id_len);
psa_status_t psa_pake_set_context(psa_pake_operation_t * operation,
                                  const uint8_t * context,
                                  size_t context_len);
psa_status_t psa_pake_output(psa_pake_operation_t * operation,
                             psa_pake_step_t step,
                             uint8_t * output,
                             size_t output_size,
                             size_t * output_length);
psa_status_t psa_pake_input(psa_pake_operation_t * operation,
                            psa_pake_step_t step,
                            const uint8_t * input,
                            size_t input_length);
psa_status_t psa_pake_get_shared_key(psa_pake_operation_t * operation,
                                     const psa_key_attributes_t * attributes,
                                     psa_key_id_t * key);
psa_status_t psa_pake_abort(psa_pake_operation_t * operation);
#define PSA_PAKE_OUTPUT_SIZE(alg, primitive, output_step) \
    /* implementation-defined value */
```

(continues on next page)

(continued from previous page)

```
#define PSA_PAKE_OUTPUT_MAX_SIZE /* implementation-defined value */
#define PSA_PAKE_INPUT_SIZE(alg, primitive, input_step) \
    /* implementation-defined value */
#define PSA_PAKE_INPUT_MAX_SIZE /* implementation-defined value */
#define PSA_ALG_JPAKE(hash_alg) /* specification-defined value */
#define PSA_ALG_IS_JPAKE(alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_HMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_CMAC(hash_alg) /* specification-defined value */
#define PSA_ALG_SPAKE2P_MATTER ((psa_algorithm_t)0x0A000609)
#define PSA_ALG_IS_SPAKE2P(alg) /* specification-defined value */
#define PSA_ALG_IS_SPAKE2P_HMAC(alg) /* specification-defined value */
#define PSA_ALG_IS_SPAKE2P_CMAC(alg) /* specification-defined value */
#define PSA_ALG_WPA3_SAE_FIXED(hash_alg) /* specification-defined value */
#define PSA_ALG_WPA3_SAE_GDH(hash_alg) /* specification-defined value */
#define PSA_ALG_IS_WPA3_SAE(alg) /* specification-defined value */
#define PSA_ALG_IS_WPA3_SAE_FIXED(alg) /* specification-defined value */
#define PSA_ALG_IS_WPA3_SAE_GDH(alg) /* specification-defined value */
#define PSA_ALG_WPA3_SAE_ANY ((psa_algorithm_t)0x0a0088ff)
psa_status_t psa_generate_random(uint8_t * output,
                                 size_t output_size);

#ifdef __cplusplus
}
#endif

#endif // PSA_CRYPTO_H
```

Appendix B: Algorithm and key type encoding

Algorithm identifiers (`psa_algorithm_t`) and key types (`psa_key_type_t`) in the Crypto API are structured integer values.

- *Algorithm identifier encoding* describes the encoding scheme for algorithm identifiers
- *Key type encoding* on page 421 describes the encoding scheme for key types

B.1 Algorithm identifier encoding

Algorithm identifiers are 32-bit integer values of the type `psa_algorithm_t`. Algorithm identifier values have the structure shown in Figure 8 on page 411.

Table 18 on page 411 describes the meaning of the bit-fields — some of the bit-fields are used in different ways by different algorithm categories.

Table 18 Bit fields in an algorithm identifier

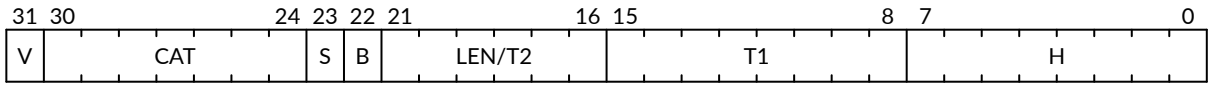

Figure 8 Encoding of `psa_algorithm_t`

Field	Bits	Description
V	[31]	Flag to indicate an implementation-defined algorithm identifier, when V=1. Algorithm identifiers defined by this specification always have V=0.
CAT	[30:24]	Algorithm category. See *Algorithm categories*.
S	[23]	For a cipher algorithm, this flag indicates a stream cipher when S=1. For a key-wrapping algorithm, this flag indicates an algorithm that accepts non-aligned input lengths when S=1. For a key-derivation algorithm, this flag indicates a key-stretching or password-hashing algorithm when S=1.
B	[22]	Flag to indicate an algorithm built on a block cipher, when B=1.
LEN/T2	[21:16]	LEN is the length of a MAC or AEAD tag, T2 is a key-agreement algorithm sub-type.
T1	[15:8]	Algorithm sub-type for most algorithm categories.
H	[7:0]	Hash algorithm sub-type, also used in any algorithm that is parameterized by a hash.

B.1.1 Algorithm categories

The CAT field in an algorithm identifier takes the values shown in Table 19 on page 412.

Table 19 Algorithm identifier categories

Algorithm category	CAT	Category details
None	`0x00`	See `PSA_ALG_NONE`
Hash	`0x02`	See *Hash algorithm encoding*
XOF	`0x0D`	See *XOF algorithm encoding* on page 413
MAC	`0x03`	See *MAC algorithm encoding* on page 414
Cipher	`0x04`	See *Cipher algorithm encoding* on page 415
AEAD	`0x05`	See *AEAD algorithm encoding* on page 415
Key wrapping	`0x0B`	See *Key-wrapping algorithm encoding* on page 416
Key derivation	`0x08`	See *Key-derivation algorithm encoding* on page 417
Asymmetric signature	`0x06`	See *Asymmetric signature algorithm encoding* on page 417
Asymmetric encryption	`0x07`	See *Asymmetric encryption algorithm encoding* on page 418
Key agreement	`0x09`	See *Key-agreement algorithm encoding* on page 419
Key encapsulation	`0x0C`	See *Key-encapsulation algorithm encoding* on page 419
PAKE	`0x0A`	See *PAKE algorithm encoding* on page 420

B.1.2 Hash algorithm encoding

The algorithm identifier for hash algorithms defined in this specification are encoded as shown in Figure 9.

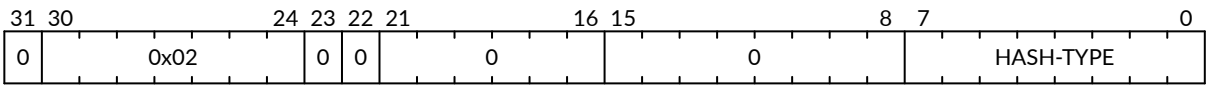

Figure 9 Hash algorithm encoding

The defined values for HASH-TYPE are shown in Table 20 on page 413.

Table 20 Hash algorithm sub-type values

Hash algorithm	HASH-TYPE	Algorithm identifier	Algorithm value
MD2	0x01	PSA_ALG_MD2	0x02000001
MD4	0x02	PSA_ALG_MD4	0x02000002
MD5	0x03	PSA_ALG_MD5	0x02000003
RIPEMD-160	0x04	PSA_ALG_RIPEMD160	0x02000004
SHA1	0x05	PSA_ALG_SHA_1	0x02000005
AES-MMO (Zigbee)	0x07	PSA_ALG_AES_MMO_ZIGBEE	0x02000007
SHA-224	0x08	PSA_ALG_SHA_224	0x02000008
SHA-256	0x09	PSA_ALG_SHA_256	0x02000009
SHA-384	0x0A	PSA_ALG_SHA_384	0x0200000A
SHA-512	0x0B	PSA_ALG_SHA_512	0x0200000B
SHA-512/224	0x0C	PSA_ALG_SHA_512_224	0x0200000C
SHA-512/256	0x0D	PSA_ALG_SHA_512_256	0x0200000D
SHA3-224	0x10	PSA_ALG_SHA3_224	0x02000010
SHA3-256	0x11	PSA_ALG_SHA3_256	0x02000011
SHA3-384	0x12	PSA_ALG_SHA3_384	0x02000012
SHA3-512	0x13	PSA_ALG_SHA3_512	0x02000013
SM3	0x14	PSA_ALG_SM3	0x02000014
SHAKE256-512	0x15	PSA_ALG_SHAKE256_512	0x02000015
Ascon-Hash256	0x19	PSA_ALG_ASCON_HASH256	0x02000019
wildcard [a]	0xFF	PSA_ALG_ANY_HASH	0x020000FF

a. The wildcard hash PSA_ALG_ANY_HASH can be used to parameterize a signature algorithm which defines a key usage policy, permitting any hash algorithm to be specified in a signature operation using the key.

B.1.3 XOF algorithm encoding

The algorithm identifier for XOF algorithms defined in this specification are encoded as shown in Figure 10.

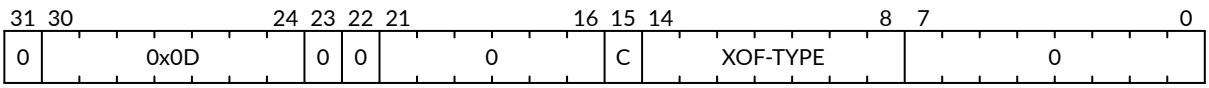

Figure 10 XOF algorithm encoding

A C value of 1 indicates that the XOF algorithm has a context parameter. The defined values for C and XOF-TYPE are shown in Table 21 on page 414.

Table 21 XOF algorithm sub-type values

XOF algorithm	C	XOF-TYPE	Algorithm identifier	Algorithm value
SHAKE128	0	0x01	PSA_ALG_SHAKE128	0x0D000100
SHAKE256	0	0x02	PSA_ALG_SHAKE256	0x0D000200
Ascon-XOF128	0	0x03	PSA_ALG_ASCON_XOF128	0x0D000300
Ascon-CXOF128	1	0x03	PSA_ALG_ASCON_CXOF128	0x0D008300

B.1.4 MAC algorithm encoding

The algorithm identifier for MAC algorithms defined in this specification are encoded as shown in Figure 11.

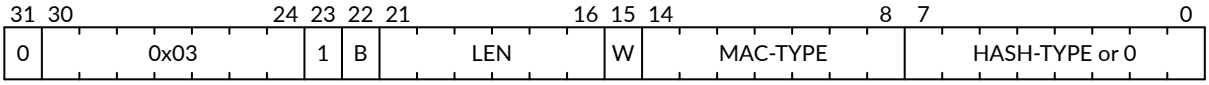

Figure 11 MAC algorithm encoding

The defined values for B and MAC-TYPE are shown in Table 22.

LEN = 0 specifies a default length output MAC, other values for LEN specify a truncated MAC.

W is a flag to indicate a wildcard permitted-algorithm policy:

- W = 0 indicates a specific MAC algorithm and MAC length.
- W = 1 indicates a wildcard key usage policy, which permits the MAC algorithm with a MAC length of at least LEN to be specified in a MAC operation using the key. LEN must not be zero.

H = HASH-TYPE (see Table 20 on page 413) for hash-based MAC algorithms, otherwise H = 0.

Table 22 MAC algorithm sub-type values

MAC algorithm	B	MAC-TYPE	Algorithm identifier	Algorithm value
HMAC	0	0x00	PSA_ALG_HMAC(hash_alg)	0x038000hh [a] [b]
CBC-MAC [c]	1	0x01	PSA_ALG_CBC_MAC	0x03c00100 [a]
CMAC [c]	1	0x02	PSA_ALG_CMAC	0x03c00200 [a]

a. This is the default algorithm identifier, specifying a standard length tag. PSA_ALG_TRUNCATED_MAC() generates identifiers with non-default LEN values. PSA_ALG_AT_LEAST_THIS_LENGTH_MAC() generates permitted-algorithm policies with W = 1.

b. hh is the HASH-TYPE for the hash algorithm, hash_alg, used to construct the MAC algorithm.

c. This is a MAC constructed using an underlying block cipher. The block cipher is determined by the key type that is provided to the MAC operation.

B.1.5 Cipher algorithm encoding

The algorithm identifier for CIPHER algorithms defined in this specification are encoded as shown in Figure 12.

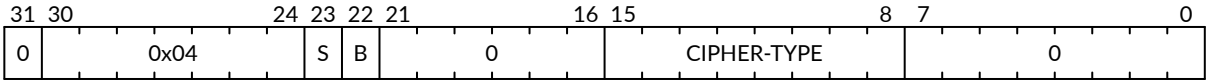

Figure 12 CIPHER algorithm encoding

The defined values for S, B, and CIPHER-TYPE are shown in Table 23.

Table 23 Cipher algorithm sub-type values

Cipher algorithm	S	B	CIPHER-TYPE	Algorithm identifier	Algorithm value
Stream cipher [a]	1	0	`0x01`	`PSA_ALG_STREAM_CIPHER`	`0x04800100`
CTR mode [b]	1	1	`0x10`	`PSA_ALG_CTR`	`0x04C01000`
CFB mode [b]	1	1	`0x11`	`PSA_ALG_CFB`	`0x04C01100`
OFB mode [b]	1	1	`0x12`	`PSA_ALG_OFB`	`0x04C01200`
CCM* with zero-length tag [b]	1	1	`0x13`	`PSA_ALG_CCM_STAR_NO_TAG`	`0x04C01300`
CCM wildcard* [c]	1	1	`0x93`	`PSA_ALG_CCM_STAR_ANY_TAG`	`0x04c09300`
XTS mode [b]	0	1	`0xFF`	`PSA_ALG_XTS`	`0x0440FF00`
CBC mode without padding [b]	0	1	`0x40`	`PSA_ALG_CBC_NO_PADDING`	`0x04404000`
CBC mode with PKCS#7 padding [b]	0	1	`0x41`	`PSA_ALG_CBC_PKCS7`	`0x04404100`
ECB mode without padding [b]	0	1	`0x44`	`PSA_ALG_ECB_NO_PADDING`	`0x04404400`

a. The stream cipher algorithm identifier `PSA_ALG_STREAM_CIPHER` is used with specific stream cipher key types, such as `PSA_KEY_TYPE_CHACHA20`.

b. This is a cipher mode of an underlying block cipher. The block cipher is determined by the key type that is provided to the cipher operation.

c. The wildcard algorithm `PSA_ALG_CCM_STAR_ANY_TAG` permits a key to be used with any CCM* algorithm: unauthenticated cipher `PSA_ALG_CCM_STAR_NO_TAG`, and AEAD algorithm `PSA_ALG_CCM`.

B.1.6 AEAD algorithm encoding

The algorithm identifier for AEAD algorithms defined in this specification are encoded as shown in Figure 13.

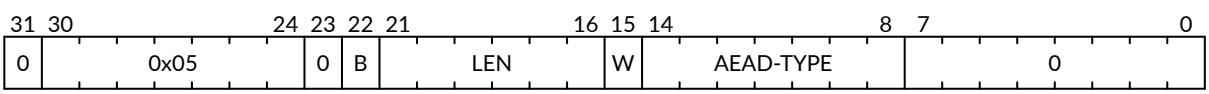

Figure 13 AEAD algorithm encoding

The defined values for B and AEAD-TYPE are shown in Table 24.

LEN = 1..31 specifies the output tag length.

W is a flag to indicate a wildcard permitted-algorithm policy:

- W = 0 indicates a specific AEAD algorithm and tag length.
- W = 1 indicates a wildcard key usage policy, which permits the AEAD algorithm with a tag length of at least LEN to be specified in an AEAD operation using the key.

Table 24 AEAD algorithm sub-type values

AEAD algorithm	B	AEAD-TYPE	Algorithm identifier	Algorithm value
CCM [a]	1	0x01	PSA_ALG_CCM	0x05500100 [b]
GCM [a]	1	0x02	PSA_ALG_GCM	0x05500200 [b]
ChaCha20-Poly1305	0	0x05	PSA_ALG_CHACHA20_POLY1305	0x05100500 [b]
XChaCha20-Poly1305	0	0x06	PSA_ALG_XCHACHA20_POLY1305	0x05100600 [b]
Ascon-AEAD128	0	0x07	PSA_ALG_ASCON_AEAD128	0x05100700 [b]

a. This is an AEAD mode of an underlying block cipher. The block cipher is determined by the key type that is provided to the AEAD operation.

b. This is the default algorithm identifier, specifying the default tag length for the algorithm. PSA_ALG_AEAD_WITH_SHORTENED_TAG() generates identifiers with alternative LEN values. PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG() generates wildcard permitted-algorithm policies with W = 1.

B.1.7 Key-wrapping algorithm encoding

The algorithm identifier for key-wrapping algorithms defined in this specification are encoded as shown in Figure 14.

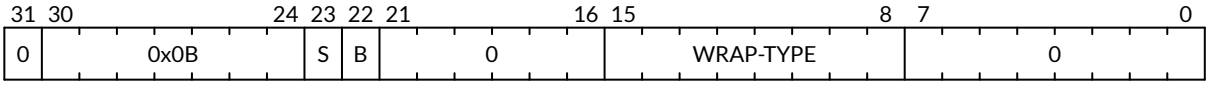

Figure 14 Key-wrapping algorithm encoding

The defined values for S, B, and WRAP-TYPE are shown in Table 25.

Table 25 Key-wrapping algorithm sub-type values

Key-wrapping algorithm	S	B	WRAP-TYPE	Algorithm identifier	Algorithm value
AES-KW	0	1	0x01	PSA_ALG_KW	0x0B400100
AES-KWP	1	1	0x02	PSA_ALG_KWP	0x0BC00200

B.1.8 Key-derivation algorithm encoding

The algorithm identifier for key-derivation algorithms defined in this specification are encoded as shown in Figure 15.

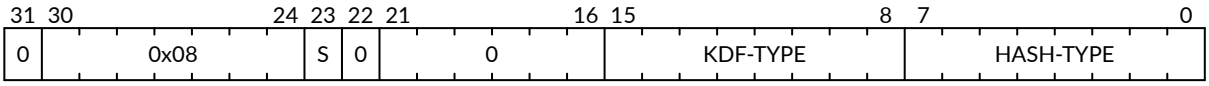

Figure 15 Key-derivation algorithm encoding

The defined values for S and KDF-TYPE are shown in Table 26.

The permitted values of HASH-TYPE (see Table 20 on page 413) depend on the specific KDF algorithm.

Table 26 Key-derivation algorithm sub-type values

Key-derivation algorithm	S	KDF-TYPE	Algorithm identifier	Algorithm value
HKDF	0	`0x01`	`PSA_ALG_HKDF(hash)`	`0x080001hh` [a]
TLS-1.2 PRF	0	`0x02`	`PSA_ALG_TLS12_PRF(hash)`	`0x080002hh` [a]
TLS-1.2 PSK-to-MasterSecret	0	`0x03`	`PSA_ALG_TLS12_PSK_TO_MS(hash)`	`0x080003hh` [a]
HKDF-Extract	0	`0x04`	`PSA_ALG_HKDF_EXTRACT(hash)`	`0x080004hh` [a]
HKDF-Expand	0	`0x05`	`PSA_ALG_HKDF_EXPAND(hash)`	`0x080005hh` [a]
TLS 1.2 ECJPAKE-to-PMS	0	`0x06`	`PSA_ALG_TLS12_ECJPAKE_TO_PMS`	`0x08000609`
SP 800-108 Counter HMAC	0	`0x07`	`PSA_ALG_SP800_108_COUNTER_HMAC(hash)`	`0x080007hh` [a]
SP 800-108 Counter CMAC	0	`0x08`	`PSA_ALG_SP800_108_COUNTER_CMAC`	`0x08000800`
PBKDF2-HMAC	1	`0x01`	`PSA_ALG_PBKDF2_HMAC(hash)`	`0x088001hh` [a]
PBKDF2-AES-CMAC-PRF-128	1	`0x02`	`PSA_ALG_PBKDF2_AES_CMAC_PRF_128`	`0x08800200`
WPA3-SAE Hash-to-element	1	`0x04`	`PSA_ALG_WPA3_SAE_H2E(hash)`	`0x088004hh` [a]

a. `hh` is the HASH-TYPE for the hash algorithm, `hash`, used to construct the key-derivation algorithm.

B.1.9 Asymmetric signature algorithm encoding

The algorithm identifier for asymmetric signature algorithms defined in this specification are encoded as shown in Figure 16.

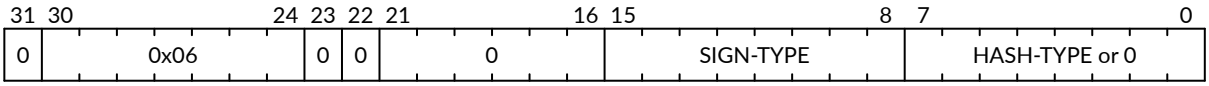

Figure 16 Asymmetric signature algorithm encoding

The defined values for SIGN-TYPE are shown in Table 27 on page 418.

H = HASH-TYPE (see Table 20 on page 413) for message signature algorithms that are parameterized by a hash algorithm, otherwise H = 0.

Table 27 Asymmetric signature algorithm sub-type values

Signature algorithm	SIGN-TYPE	Algorithm identifier	Algorithm value
RSA PKCS#1 v1.5	0x02	PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg)	0x060002hh [a]
RSA PKCS#1 v1.5 no hash [b]	0x02	PSA_ALG_RSA_PKCS1V15_SIGN_RAW	0x06000200
RSA PSS	0x03	PSA_ALG_RSA_PSS(hash_alg)	0x060003hh [a]
RSA PSS any salt length	0x13	PSA_ALG_RSA_PSS_ANY_SALT(hash_alg)	0x060013hh [a]
Randomized ECDSA	0x06	PSA_ALG_ECDSA(hash_alg)	0x060006hh [a]
Randomized ECDSA no hash [b]	0x06	PSA_ALG_ECDSA_ANY	0x06000600
Deterministic ECDSA	0x07	PSA_ALG_DETERMINISTIC_ECDSA(hash_alg)	0x060007hh [a]
PureEdDSA without context	0x08	PSA_ALG_PURE_EDDSA	0x06000800
HashEdDSA	0x09	PSA_ALG_ED25519PH and PSA_ALG_ED448PH	0x060009hh [c]
PureEdDSA with context	0x0a	PSA_ALG_EDDSA_CTX	0x06000a00

a. hh is the HASH-TYPE for the hash algorithm, hash_alg, used to construct the signature algorithm.

b. Asymmetric signature algorithms without hashing can only be used with psa_sign_hash() and psa_verify_hash().

c. The HASH-TYPE for HashEdDSA is determined by the curve. SHA-512 is used for Ed25519ph, and the first 64 bytes of output from SHAKE256 is used for Ed448ph.

B.1.10 Asymmetric encryption algorithm encoding

The algorithm identifier for asymmetric encryption algorithms defined in this specification are encoded as shown in Figure 17.

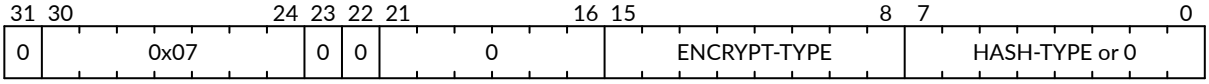

Figure 17 Asymmetric encryption algorithm encoding

The defined values for ENCRYPT-TYPE are shown in Table 28.

H = HASH-TYPE (see Table 20 on page 413) for asymmetric encryption algorithms that are parameterized by a hash algorithm, otherwise H = 0.

Table 28 Asymmetric encryption algorithm sub-type values

Asymmetric encryption algorithm	ENCRYPT-TYPE	Algorithm identifier	Algorithm value
RSA PKCS#1 v1.5	0x02	PSA_ALG_RSA_PKCS1V15_CRYPT	0x07000200
RSA OAEP	0x03	PSA_ALG_RSA_OAEP(hash_alg)	0x070003hh [a]

a. `hh` is the HASH-TYPE for the hash algorithm, `hash_alg`, used to construct the encryption algorithm.

B.1.11 Key-agreement algorithm encoding

A key-agreement algorithm identifier can either be for the standalone key-agreement algorithm, or for a combined key-agreement with key-derivation algorithm. The former can only be used with `psa_key_agreement()` and `psa_raw_key_agreement()`, while the latter are used with `psa_key_derivation_key_agreement()`.

The algorithm identifier for standalone key-agreement algorithms defined in this specification are encoded as shown in Figure 18.

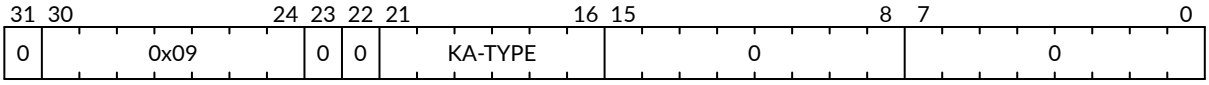

Figure 18 Standalone key-agreement algorithm encoding

The defined values for KA-TYPE are shown in Table 29.

Table 29 Key-agreement algorithm sub-type values

Key-agreement algorithm	KA-TYPE	Algorithm identifier	Algorithm value
FFDH	`0x01`	`PSA_ALG_FFDH`	`0x09010000`
ECDH	`0x02`	`PSA_ALG_ECDH`	`0x09020000`

A combined key agreement is constructed by a bitwise OR of the standalone key-agreement algorithm identifier and the key-derivation algorithm identifier. This operation is provided by the `PSA_ALG_KEY_AGREEMENT()` macro.

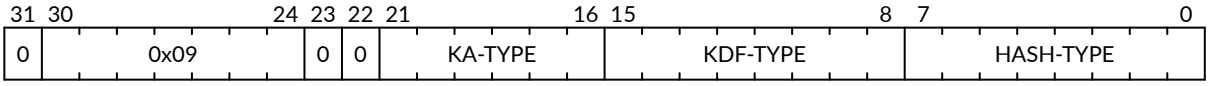

Figure 19 Combined key-agreement algorithm encoding

The underlying standalone key-agreement algorithm can be extracted from the KA-TYPE field, and the key-derivation algorithm from the KDF-TYPE and HASH-TYPE fields.

B.1.12 Key-encapsulation algorithm encoding

The algorithm identifier for key-encapsulation algorithms defined in this specification are encoded as shown in Figure 20 on page 420.

The defined values for ENCAPS-TYPE are shown in Table 30 on page 420.

Table 30 Encapsulation algorithm sub-type values

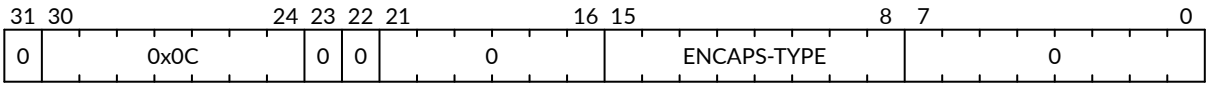

Figure 20 Encapsulation algorithm encoding

Encapsulation algorithm	ENCAPS-TYPE	Algorithm identifier	Algorithm value
ECIES (SEC1)	`0x01`	`PSA_ALG_ECIES_SEC1`	`0x0C000100`

B.1.13 PAKE algorithm encoding

The algorithm identifier for PAKE algorithms defined in this specification are encoded as shown in Figure 21.

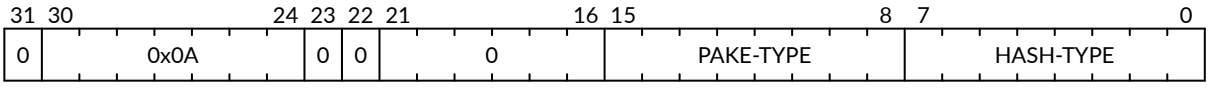

Figure 21 PAKE algorithm encoding

The defined values for PAKE-TYPE are shown in Table 31.

The permitted values of HASH-TYPE (see Table 20 on page 413) depend on the specific PAKE algorithm.

Table 31 PAKE algorithm sub-type values

PAKE algorithm	PAKE-TYPE	Algorithm identifier	Algorithm value
J-PAKE	`0x01`	`PSA_ALG_JPAKE(hash)`	`0x0A0001hh` [a]
SPAKE2+ with HMAC	`0x04`	`PSA_ALG_SPAKE2P_HMAC(hash)`	`0x0A0004hh` [a]
SPAKE2+ with CMAC	`0x05`	`PSA_ALG_SPAKE2P_CMAC(hash)`	`0x0A0005hh` [a]
SPAKE2+ for Matter	`0x06`	`PSA_ALG_SPAKE2P_MATTER`	`0x0A000609`
WPA3-SAE	`0x08`	`PSA_ALG_WPA3_SAE_FIXED(hash)`	`0x0A0008hh` [a]
WPA3-SAE (GDH)	`0x09`	`PSA_ALG_WPA3_SAE_GDH(hash)`	`0x0A0009hh` [a]
WPA3-SAE wildcard [b c]	`0x88`	`PSA_ALG_WPA3_SAE_ANY`	`0x0A0088FF`

a. `hh` is the HASH-TYPE for the hash algorithm, `hash`, used to construct the key-derivation algorithm.

b. The wildcard algorithm `PSA_ALG_WPA3_SAE_ANY` permits a password key to be used for any WPA3-SAE cipher suite with the `PSA_ALG_WPA3_SAE_H2E` key-derivation algorithm, and with the `PSA_ALG_WPA3_SAE_FIXED` PAKE algorithm.

c. The wildcard algorithm `PSA_ALG_WPA3_SAE_ANY` permits a WPA3-SAE password token key to be used for both the `PSA_ALG_WPA3_SAE_FIXED` and `PSA_ALG_WPA3_SAE_GDH` PAKE algorithms.

B.2 Key type encoding

Key types are 16-bit integer values of the type `psa_key_type_t`. Key type values have the structure shown in Figure 22.

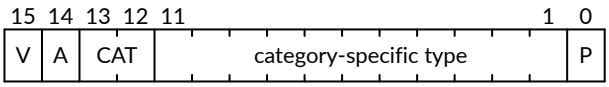

Figure 22 Encoding of `psa_key_type_t`

Table 32 describes the meaning of the bit-fields — some of bit-fields are used in different ways by different key type categories.

Table 32 Bit fields in a key type

Field	Bits	Description
V	[15]	Flag to indicate an implementation-defined key type, when V=1. Key types defined by this specification always have V=0.
A	[14]	Flag to indicate an asymmetric key type, when A=1.
CAT	[13:12]	Key type category. See *Key type categories*.
category-specific type	[11:1]	The meaning of this field is specific to each key category.
P	[0]	Parity bit. Valid key type values have even parity.

B.2.1 Key type categories

The A and CAT fields in a key type take the values shown in Table 33.

Table 33 Key type categories

Key type category	A	CAT	Category details
None	0	0	See `PSA_KEY_TYPE_NONE`
Raw data	0	1	See *Raw key encoding*
Symmetric key	0	2	See *Symmetric key encoding* on page 422
Structured key	0	3	See *Structured key encoding* on page 423
Asymmetric public key	1	0	See *Asymmetric key encoding* on page 424
Asymmetric key pair	1	3	See *Asymmetric key encoding* on page 424

B.2.2 Raw key encoding

The key type for raw keys defined in this specification are encoded as shown in Figure 23 on page 422.

The defined values for RAW-TYPE, SUB-TYPE, and P are shown in Table 34 on page 422.

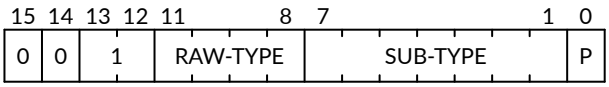

Figure 23 Raw key encoding

Table 34 Raw key sub-type values

Raw key type	RAW-TYPE	SUB-TYPE	P	Key type	Key type value
Raw data	0	0	1	PSA_KEY_TYPE_RAW_DATA	0x1001
HMAC	1	0	0	PSA_KEY_TYPE_HMAC	0x1100
Derivation secret	2	0	0	PSA_KEY_TYPE_DERIVE	0x1200
Password	2	1	1	PSA_KEY_TYPE_PASSWORD	0x1203
Password hash	2	2	1	PSA_KEY_TYPE_PASSWORD_HASH	0x1205
Derivation pepper	2	3	0	PSA_KEY_TYPE_PEPPER	0x1206

B.2.3 Symmetric key encoding

The key type for symmetric keys defined in this specification are encoded as shown in Figure 24.

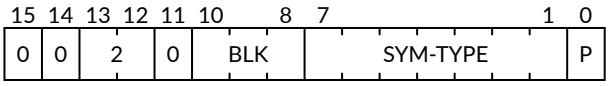

Figure 24 Symmetric key encoding

For block-based cipher keys, the block size for the cipher algorithm is 2^{BLK}.

The defined values for BLK, SYM-TYPE and P are shown in Table 35.

Table 35 Symmetric key sub-type values

Symmetric key type	BLK	SYM-TYPE	P	Key type	Key type value
ARC4	0	1	0	PSA_KEY_TYPE_ARC4	0x2002
ChaCha20	0	2	0	PSA_KEY_TYPE_CHACHA20	0x2004
XChaCha20	0	3	1	PSA_KEY_TYPE_XCHACHA20	0x2007
Ascon	0	4	0	PSA_KEY_TYPE_ASCON	0x2008
DES	3	0	1	PSA_KEY_TYPE_DES	0x2301
AES	4	0	0	PSA_KEY_TYPE_AES	0x2400
CAMELLIA	4	1	1	PSA_KEY_TYPE_CAMELLIA	0x2403
SM4	4	2	1	PSA_KEY_TYPE_SM4	0x2405
ARIA	4	3	0	PSA_KEY_TYPE_ARIA	0x2406

B.2.4 Structured key encoding

The key type for structured keys defined in this specification are encoded as shown in Figure 25.

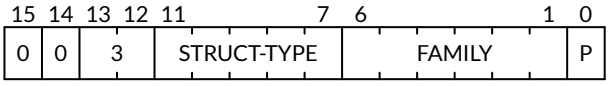

Figure 25 Encoding of structured keys

The defined values for STRUCT-TYPE are shown in Table 36.

The defined values for FAMILY depend on the STRUCT-TYPE value. See the details for each structured key sub-type.

Table 36 Structured key sub-type values

Structured key type	STRUCT-TYPE	Details
WPA3-SAE password token	5, 6	See *WPA3-SAE password token encoding*

WPA3-SAE password token encoding

WPA3-SAE is defined to use either elliptic curve or finite field groups. These use distinct STRUCT-TYPE values, and use the same FAMILY values as elliptic curve and finite field Diffie-Hellman key types.

WPA3-SAE password tokens using elliptic curves

The key type for WPA3-SAE password tokens using elliptic curves defined in this specification are encoded as shown in Figure 26.

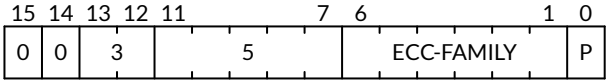

Figure 26 Encoding of WPA3-SAE password token using elliptic curves

The defined values for ECC-FAMILY and P are shown in Table 37.

Table 37 WPA3-SAE password token ECC family values

WPA3-SAE suite	ECC-FAMILY	P	ECC family [a]	Key value
SECP R1	0x09	0	`PSA_ECC_FAMILY_SECP_R1`	`0x3292`
Brainpool-P R1	0x18	0	`PSA_ECC_FAMILY_BRAINPOOL_P_R1`	`0x32b0`

a. The elliptic curve family values defined in the API also include the parity bit. The password token key type value is constructed from the elliptic curve family using `PSA_KEY_TYPE_WPA3_SAE_ECC(family)`.

WPA3-SAE password tokens using finite fields

The key type for WPA3-SAE password tokens using finite fields defined in this specification are encoded as shown in Figure 27.

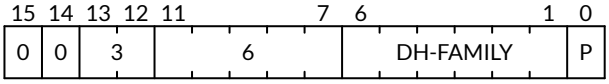

Figure 27 Encoding of WPA3-SAE password token using finite fields

The defined values for DH-FAMILY and P are shown in Table 38.

RFC3526 defines a set of FF groups that are recommended for use with WPA3-SAE (those with primes >=3072 bits)

Table 38 WPA3-SAE password token finite field Diffie-Hellman family values

WPA3-SAE suite	DH-FAMILY	P	DH family [a]	Key value
RFC3526	0x02	1	`PSA_DH_FAMILY_RFC3526`	`0x3305`

a. The finite field Diffie Hellman family values defined in the API also include the parity bit. The password token key type value is constructed from the finite field Diffie Hellman family using `PSA_KEY_TYPE_WPA3_SAE_DH(family)`.

B.2.5 Asymmetric key encoding

The key type for asymmetric keys defined in this specification are encoded as shown in Figure 28.

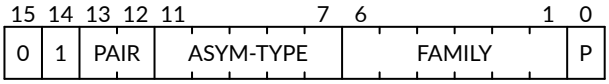

Figure 28 Asymmetric key encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for ASYM-TYPE are shown in Table 39.

The defined values for FAMILY depend on the ASYM-TYPE value. See the details for each asymmetric key sub-type.

Table 39 Asymmetric key sub-type values

Asymmetric key type	ASYM-TYPE	Details
Non-parameterized	0	See *Non-parameterized asymmetric key encoding* on page 425
Elliptic Curve	2	See *Elliptic curve key encoding* on page 425
Diffie-Hellman	4	See *Finite field Diffie Hellman key encoding* on page 426
SPAKE2+	8	See *SPAKE2+ key encoding* on page 427

Non-parameterized asymmetric key encoding

The key type for non-parameterized asymmetric keys defined in this specification are encoded as shown in Figure 29.

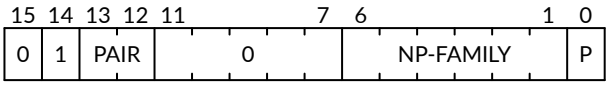

Figure 29 Non-parameterized asymmetric keys encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for NP-FAMILY and P are shown in Table 40.

Table 40 Non-parameterized asymmetric key family values

Key family	Public/pair	PAIR	NP-FAMILY	P	Key type	Key value
RSA	Public key	0	0	1	`PSA_KEY_TYPE_RSA_PUBLIC_KEY`	`0x4001`
	Key pair	3	0	1	`PSA_KEY_TYPE_RSA_KEY_PAIR`	`0x7001`

Elliptic curve key encoding

The key type for elliptic curve keys defined in this specification are encoded as shown in Figure 30.

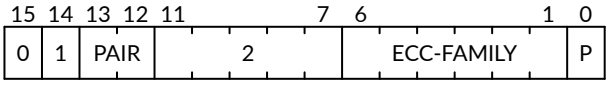

Figure 30 Elliptic curve key encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for ECC-FAMILY and P are shown in Table 41 on page 426.

Table 41 ECC key family values

ECC key family	ECC-FAMILY	P	ECC family [a]	Public-key value	Key-pair value
SECP K1	0x0B	1	PSA_ECC_FAMILY_SECP_K1	0x4117	0x7117
SECP R1	0x09	0	PSA_ECC_FAMILY_SECP_R1	0x4112	0x7112
SECP R2	0x0D	1	PSA_ECC_FAMILY_SECP_R2	0x411B	0x711B
SECT K1	0x13	1	PSA_ECC_FAMILY_SECT_K1	0x4127	0x7127
SECT R1	0x11	0	PSA_ECC_FAMILY_SECT_R1	0x4122	0x7122
SECT R2	0x15	1	PSA_ECC_FAMILY_SECT_R2	0x412B	0x712B
Brainpool-P R1	0x18	0	PSA_ECC_FAMILY_BRAINPOOL_P_R1	0x4130	0x7130
FRP	0x19	1	PSA_ECC_FAMILY_FRP	0x4133	0x7133
Montgomery	0x20	1	PSA_ECC_FAMILY_MONTGOMERY	0x4141	0x7141
Twisted Edwards	0x21	0	PSA_ECC_FAMILY_TWISTED_EDWARDS	0x4142	0x7142

a. The elliptic curve family values defined in the API also include the parity bit. The key type value is constructed from the elliptic curve family using either PSA_KEY_TYPE_ECC_PUBLIC_KEY(family) or PSA_KEY_TYPE_ECC_KEY_PAIR(family) as required.

Finite field Diffie Hellman key encoding

The key type for finite field Diffie Hellman keys defined in this specification are encoded as shown in Figure 31.

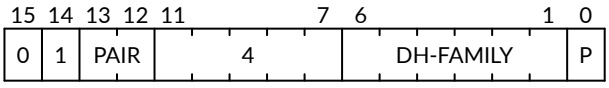

Figure 31 Finite field Diffie Hellman key encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for DH-FAMILY and P are shown in Table 42.

Table 42 Finite field Diffie Hellman key group values

DH key group	DH-FAMILY	P	DH family [a]	Public-key value	Key-pair value
RFC7919	0x01	1	PSA_DH_FAMILY_RFC7919	0x4203	0x7203

a. The finite field Diffie Hellman group family values defined in the API also include the parity bit. The key type value is constructed from the finite field Diffie Hellman family using either PSA_KEY_TYPE_DH_PUBLIC_KEY(family) or PSA_KEY_TYPE_DH_KEY_PAIR(family) as required.

SPAKE2+ key encoding

The key type for SPAKE2+ keys defined in this specification are encoded as shown in Figure 32.

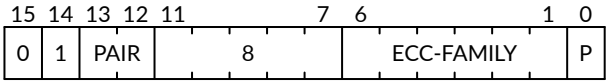

Figure 32 SPAKE2+ key encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for ECC-FAMILY and P are shown in Table 43.

Table 43 SPAKE2+ key family values

SPAKE2+ group	ECC-FAMILY	P	ECC family [a]	Public-key value	Key-pair value
SECP R1	0x09	0	`PSA_ECC_FAMILY_SECP_R1`	`0x4412`	`0x7412`
Twisted Edwards	0x21	0	`PSA_ECC_FAMILY_TWISTED_EDWARDS`	`0x4442`	`0x7442`

a. The elliptic curve family values defined in the API also include the parity bit. The key type value is constructed from the elliptic curve family using either `PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(family)` or `PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(family)` as required.

Appendix C: 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.

C.1 Algorithm macros

```
#define PSA_ALG_AEAD_WITH_DEFAULT_LENGTH_TAG(aead_alg) \
    ((((aead_alg) & ~0x003f8000) == 0x05400100) ? PSA_ALG_CCM : \
     (((aead_alg) & ~0x003f8000) == 0x05400200) ? PSA_ALG_GCM : \
     (((aead_alg) & ~0x003f8000) == 0x05000500) ? PSA_ALG_CHACHA20_POLY1305 : \
     PSA_ALG_NONE)

#define PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG(aead_alg, min_tag_length) \
    ( PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, min_tag_length) | 0x00008000 )

#define PSA_ALG_AEAD_WITH_SHORTENED_TAG(aead_alg, tag_length) \
    ((psa_algorithm_t) (((aead_alg) & ~0x003f8000) | (((tag_length) & 0x3f) << 16)))

#define PSA_ALG_AT_LEAST_THIS_LENGTH_MAC(mac_alg, min_mac_length) \
    ( PSA_ALG_TRUNCATED_MAC(mac_alg, min_mac_length) | 0x00008000 )

#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) & ~0x003f8000))

#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_HKDF_EXPAND(hash_alg) \
    ((psa_algorithm_t) (0x08000500 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_HKDF_EXTRACT(hash_alg) \
    ((psa_algorithm_t) (0x08000400 | ((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)
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) \
    (((alg) & 0x7fc00000) == 0x03c00000)

#define PSA_ALG_IS_CIPHER(alg) \
    (((alg) & 0x7f000000) == 0x04000000)

#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) || PSA_ALG_IS_HASH_EDDSA(alg))

#define PSA_ALG_IS_HASH_EDDSA(alg) \
    (((alg) & ~0x000000ff) == 0x06000900)

#define PSA_ALG_IS_HKDF(alg) \
    (((alg) & ~0x000000ff) == 0x08000100)

#define PSA_ALG_IS_HKDF_EXPAND(alg) \
    (((alg) & ~0x000000ff) == 0x08000500)

#define PSA_ALG_IS_HKDF_EXTRACT(alg) \
    (((alg) & ~0x000000ff) == 0x08000400)

#define PSA_ALG_IS_HMAC(alg) \
    (((alg) & 0x7fc0ff00) == 0x03800000)

#define PSA_ALG_IS_JPAKE(alg) \
    (((alg) & ~0x000000ff) == 0x0a000100)

#define PSA_ALG_IS_KEY_AGREEMENT(alg) \
    (((alg) & 0x7f000000) == 0x09000000)

#define PSA_ALG_IS_KEY_DERIVATION(alg) \
```

(continues on next page)

(continued from previous page)

```
    (((alg) & 0x7f000000) == 0x08000000)

#define PSA_ALG_IS_KEY_DERIVATION_STRETCHING(alg) \
    (((alg) & 0x7f800000) == 0x08800000)

#define PSA_ALG_IS_KEY_ENCAPSULATION(alg) \
    (((alg) & 0x7f000000) == 0x0c000000)

#define PSA_ALG_IS_KEY_WRAP(alg) \
    (((alg) & 0x7f000000) == 0x0b000000)

#define PSA_ALG_IS_MAC(alg) \
    (((alg) & 0x7f000000) == 0x03000000)

#define PSA_ALG_IS_PAKE(alg) \
    (((alg) & 0x7f000000) == 0x0a000000)

#define PSA_ALG_IS_PBKDF2_HMAC(alg) \
    (((alg) & ~0x000000ff) == 0x08800100)

#define PSA_ALG_IS_RANDOMIZED_ECDSA(alg) \
    (((alg) & ~0x000000ff) == 0x06000600)

#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) & ~0x000010ff) == 0x06000300)

#define PSA_ALG_IS_RSA_PSS_ANY_SALT(alg) \
    (((alg) & ~0x000000ff) == 0x06001300)

#define PSA_ALG_IS_RSA_PSS_STANDARD_SALT(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) && \
     (alg) != PSA_ALG_PURE_EDDSA && (alg) != PSA_ALG_EDDSA_CTX)

#define PSA_ALG_IS_SIGN_MESSAGE(alg) \
    (PSA_ALG_IS_SIGN(alg) && \
     (alg) != PSA_ALG_ECDSA_ANY && (alg) != PSA_ALG_RSA_PKCS1V15_SIGN_RAW)
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_IS_SP800_108_COUNTER_HMAC(alg) \
    (((alg) & ~0x000000ff) == 0x08000700)

#define PSA_ALG_IS_SPAKE2P(alg) \
    (((alg) & ~0x000003ff) == 0x0a000400)

#define PSA_ALG_IS_SPAKE2P_CMAC(alg) \
    (((alg) & ~0x000000ff) == 0x0a000500)

#define PSA_ALG_IS_SPAKE2P_HMAC(alg) \
    (((alg) & ~0x000000ff) == 0x0a000400)

#define PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg) \
    (((alg) & 0x7f00ffff) == 0x09000000)

#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_ANY_HASH || \
     ((alg) & 0x7f008000) == 0x03008000 || \
     ((alg) & 0x7f008000) == 0x05008000 || \
     (alg) == PSA_ALG_CCM_STAR_ANY_TAG)

#define PSA_ALG_IS_WPA3_SAE(alg) \
    (((alg) & ~0x000001ff) == 0x0a000800)

#define PSA_ALG_IS_WPA3_SAE_FIXED(alg) \
    (((alg) & ~0x000000ff) == 0x0a000800)

#define PSA_ALG_IS_WPA3_SAE_GDH(alg) \
    (((alg) & ~0x000000ff) == 0x0a000900)

#define PSA_ALG_IS_WPA3_SAE_H2E(alg) \
    (((alg) & ~0x000000ff) == 0x08800400)

#define PSA_ALG_IS_XOF(alg) \
    (((alg) & 0x7f000000) == 0x0D000000)

#define PSA_ALG_JPAKE(hash_alg) \
    ((psa_algorithm_t) (0x0a000100 | ((hash_alg) & 0x000000ff)))
```

(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) & 0xff7f0000))

#define PSA_ALG_KEY_AGREEMENT_GET_KDF(alg) \
    ((psa_algorithm_t) ((alg) & 0xfe80ffff))

#define PSA_ALG_PBKDF2_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x08800100 | ((hash_alg) & 0x000000ff)))

#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_RSA_PSS_ANY_SALT(hash_alg) \
    ((psa_algorithm_t) (0x06001300 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_SP800_108_COUNTER_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x08000700 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_SPAKE2P_CMAC(hash_alg) \
    ((psa_algorithm_t) (0x0a000500 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_SPAKE2P_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x0a000400 | ((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) & ~0x003f8000) | (((mac_length) & 0x3f) << 16)))

#define PSA_ALG_WPA3_SAE_FIXED(hash_alg) \
    ((psa_algorithm_t) (0x0a000800 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_WPA3_SAE_GDH(hash_alg) \
    ((psa_algorithm_t) (0x0a000900 | ((hash_alg) & 0x000000ff)))
```

(continues on next page)

(continued from previous page)

```
#define PSA_ALG_WPA3_SAE_H2E(hash_alg) \
    ((psa_algorithm_t) (0x08800400 | ((hash_alg) & 0x000000ff)))

#define PSA_ALG_XOF_HAS_CONTEXT(alg) \
    (((alg) & 0x00008000) != 0)

#define PSA_PAKE_PRIMITIVE(pake_type, pake_family, pake_bits) \
    ((pake_bits & 0xFFFF) != pake_bits) ? 0 :                 \
    ((psa_pake_primitive_t) (((pake_type) << 24 |             \
            (pake_family) << 16) | (pake_bits)))

#define PSA_PAKE_PRIMITIVE_GET_BITS(pake_primitive) \
    ((size_t)(pake_primitive & 0xFFFF))

#define PSA_PAKE_PRIMITIVE_GET_FAMILY(pake_primitive) \
    ((psa_pake_family_t)((pake_primitive >> 16) & 0xFF))

#define PSA_PAKE_PRIMITIVE_GET_TYPE(pake_primitive) \
    ((psa_pake_primitive_type_t)((pake_primitive >> 24) & 0xFF))
```

C.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) & 0x007f))

#define PSA_KEY_TYPE_DH_KEY_PAIR(group) \
    ((psa_key_type_t) (0x7200 | ((group) & 0x007f)))

#define PSA_KEY_TYPE_DH_PUBLIC_KEY(group) \
    ((psa_key_type_t) (0x4200 | ((group) & 0x007f)))

#define PSA_KEY_TYPE_ECC_GET_FAMILY(type) \
    ((psa_ecc_family_t) ((type) & 0x007f))

#define PSA_KEY_TYPE_ECC_KEY_PAIR(curve) \
    ((psa_key_type_t) (0x7100 | ((curve) & 0x007f)))

#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) \
    ((psa_key_type_t) (0x4100 | ((curve) & 0x007f)))

#define PSA_KEY_TYPE_IS_ASYMMETRIC(type) \
    (((type) & 0x4000) == 0x4000)
```

(continues on next page)

(continued from previous page)

```
#define PSA_KEY_TYPE_IS_DH(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff80) == 0x4200)

#define PSA_KEY_TYPE_IS_DH_KEY_PAIR(type) \
    (((type) & 0xff80) == 0x7200)

#define PSA_KEY_TYPE_IS_DH_PUBLIC_KEY(type) \
    (((type) & 0xff80) == 0x4200)

#define PSA_KEY_TYPE_IS_ECC(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff80) == 0x4100)

#define PSA_KEY_TYPE_IS_ECC_KEY_PAIR(type) \
    (((type) & 0xff80) == 0x7100)

#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(type) \
    (((type) & 0xff80) == 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_SPAKE2P(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff80) == 0x4400)

#define PSA_KEY_TYPE_IS_SPAKE2P_KEY_PAIR(type) \
    (((type) & 0xff80) == 0x7400)

#define PSA_KEY_TYPE_IS_SPAKE2P_PUBLIC_KEY(type) \
    (((type) & 0xff80) == 0x4400)

#define PSA_KEY_TYPE_IS_UNSTRUCTURED(type) \
    (((type) & 0x7000) == 0x1000 || ((type) & 0x7000) == 0x2000)

#define PSA_KEY_TYPE_IS_WPA3_SAE_DH(type) \
    (((type) & 0xff80) == 0x3300)

#define PSA_KEY_TYPE_IS_WPA3_SAE_ECC(type) \
    (((type) & 0xff80) == 0x3280)

#define PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type) \
    ((psa_key_type_t) ((type) | 0x3000))
```

(continues on next page)

(continued from previous page)

```
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) \
    ((psa_key_type_t) ((type) & ~0x3000))

#define PSA_KEY_TYPE_SPAKE2P_GET_FAMILY(type) \
    ((psa_ecc_family_t) ((type) & 0x007f))

#define PSA_KEY_TYPE_SPAKE2P_KEY_PAIR(curve) \
    ((psa_key_type_t) (0x7400 | ((curve) & 0x007f)))

#define PSA_KEY_TYPE_SPAKE2P_PUBLIC_KEY(curve) \
    ((psa_key_type_t) (0x4400 | ((curve) & 0x007f)))

#define PSA_KEY_TYPE_WPA3_SAE_DH_GET_FAMILY(type) \
    ((psa_dh_family_t) ((type) & 0x007f))

#define PSA_KEY_TYPE_WPA3_SAE_DH(family) \
    ((psa_key_type_t) (0x3300 | ((family) & 0x007f)))

#define PSA_KEY_TYPE_WPA3_SAE_ECC_GET_FAMILY(type) \
    ((psa_ecc_family_t) ((type) & 0x007f))

#define PSA_KEY_TYPE_WPA3_SAE_ECC(curve) \
    ((psa_key_type_t) (0x3280 | ((curve) & 0x007f)))
```

C.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_224 || (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 || \
        (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_224 || (alg)==PSA_ALG_SHA_512_256 ? 16 : \
     0)

#define PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) \
    (PSA_HASH_SUSPEND_ALGORITHM_FIELD_LENGTH + \
```

(continues on next page)

(continued from previous page)

```
    PSA_HASH_SUSPEND_INPUT_LENGTH_FIELD_LENGTH(alg) + \
    PSA_HASH_SUSPEND_HASH_STATE_FIELD_LENGTH(alg) + \
    PSA_HASH_BLOCK_LENGTH(alg) - 1)
```

Appendix D: Security Risk Assessment

This Security Risk Assessment (SRA) analyses the security of the Crypto API itself, not of any specific implementation of the API, or any specific use of the API. However, the security of an implementation of the Crypto API depends on the implementation design, the capabilities of the system in which it is deployed, and the need to address some of the threats identified in this assessment.

To enable the Crypto API to be suitable for a wider range of security use cases, this SRA considers a broad range of adversarial models and threats to the application and the implementation, as well as to the API.

This approach allows the assessment to identify API design requirements that affect the ability for an implementation to mitigate threats that do not directly attack the API.

The scope is described in *Adversarial models* on page 439.

D.1 Architecture

D.1.1 System definition

Figure 33 shows the Crypto API as the defined interface that an Application uses to interact with the Cryptoprocessor.

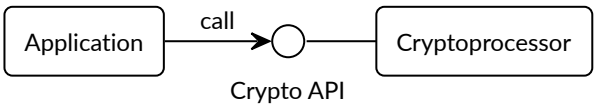

Figure 33 Crypto API

Assumptions, constraints, and interacting entities

This SRA makes the following assumptions about the Crypto API design:

- The API does not provide arguments that identify the caller, because they can be spoofed easily, and cannot be relied upon. It is assumed that the implementation of the API can determine the caller identity, where this is required. See *Optional isolation* on page 21.
- The API does not prevent the use of mitigations that are required by an implementation of the API. See *Implementation remediations* on page 447.
- The API follows best-practices for C interface design, reducing the risk of exploitable errors in the application and implementation code. See *Ease of use* on page 22.

Trust boundaries and information flow

The Crypto API is the interface available to the programmer, and is the main attack surface that is analyzed here. However, to ensure that the API enables the mitigation of other threats to an implementation, we also consider the system context in which the Crypto API is used.

Figure 34 shows the data flow for a typical application usage of the Crypto API, for example, to exchange ciphertext with an external system, or for at rest protection in system non-volatile storage. The Application uses the Crypto API to interact with the Cryptoprocessor. The Cryptoprocessor stores persistent keys in a Key Store.

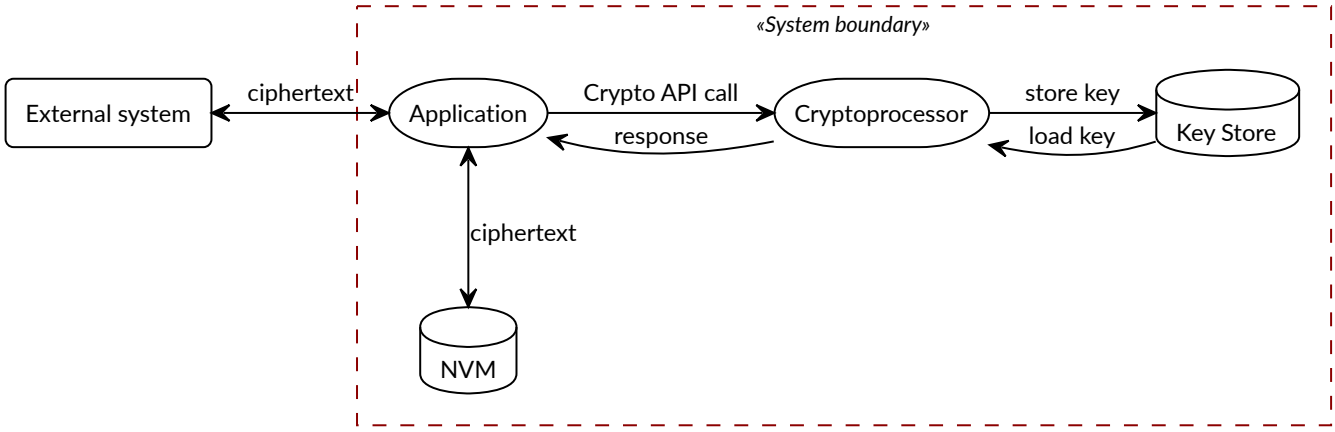

Figure 34 Crypto API dataflow diagram for an implementation with no isolation

For some adversarial models, *Cryptoprocessor isolation* or *Caller isolation* is required in the implementation to achieve the security goals. See *Security goals* on page 439, and remediations R.1 and R.2 in *Implementation remediations* on page 447.

The Cryptoprocessor can optionally include a trust boundary within its implementation of the API. The trust boundary shown in Figure 35 on page 438 corresponds to Cryptoprocessor isolation. The Cryptoprocessor boundary protects the confidentiality and integrity of the Cryptoprocessor and Key Store state from system components that are outside of the boundary.

If the implementation supports multiple, independent client Applications within the system, each Application has its own view of the Cryptoprocessor and key store. The additional trust boundaries required for a caller isolated implementation are shown in Figure 36 on page 438. The Application boundary restricts the capabilities of the Application, and protects the confidentiality and integrity of system state from the Application.

D.1.2 Assets and stakeholders

1. Cryptographic keys and key-related assets. This includes the key properties, such as the key type, identity and policies.

   Stakeholders can include the SiP, the OEM, the system or application owner. Owners of a key need to be able to use the key for cryptographic operations, such as encryption or signature, and where permitted, delete, copy or extract the key.

   Disclosure of the cryptographic key material to an attacker defeats the protection that the use of cryptography provides. Modification of cryptographic key material or key properties by an attacker has the same end result. These allow an attacker access to the assets that are protected by the key.

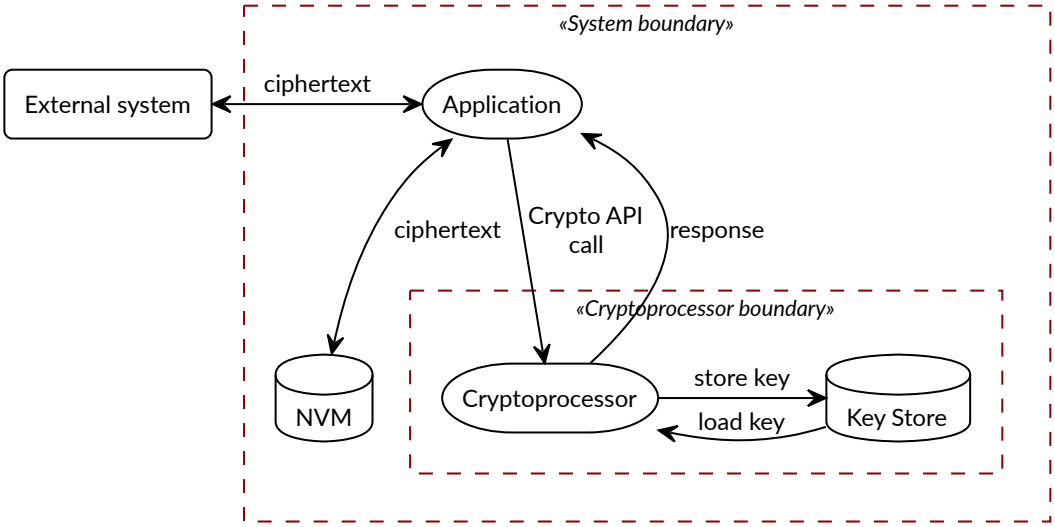

Figure 35 Crypto API dataflow diagram for an implementation with cryptoprocessor isolation

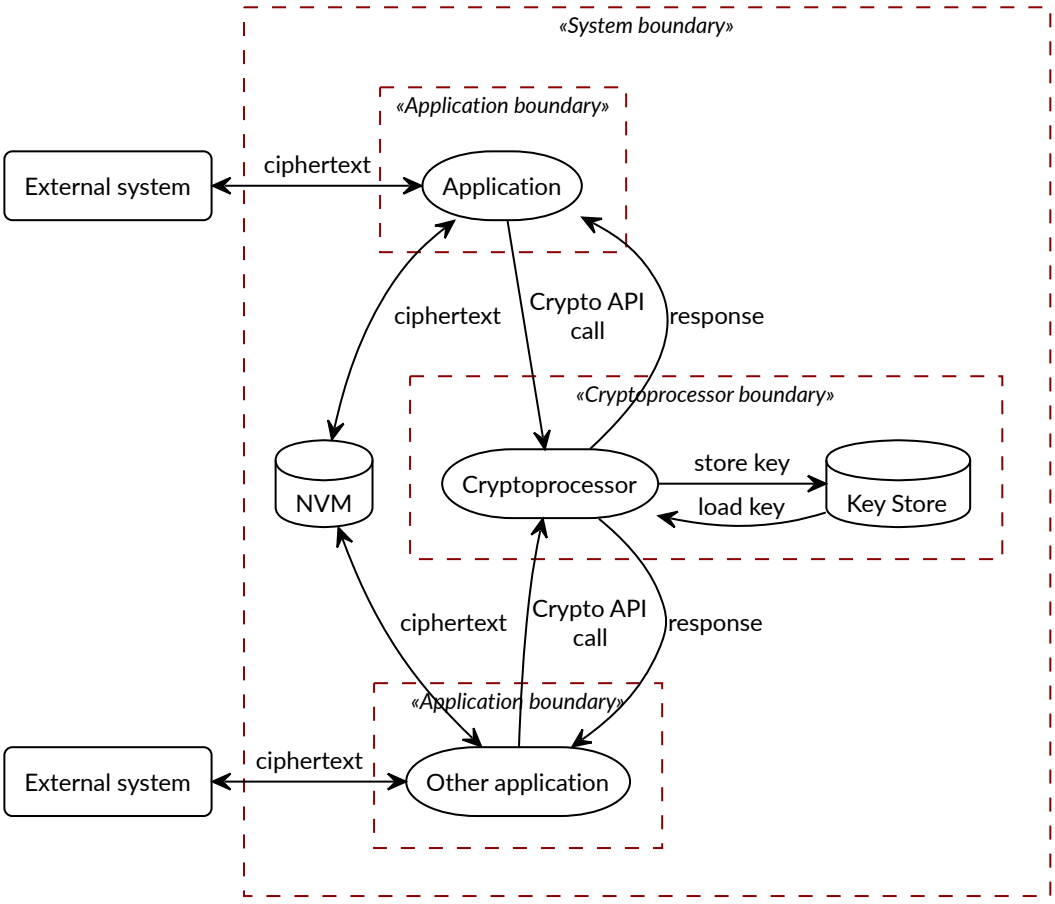

Figure 36 Crypto API dataflow diagram for an implementation with caller isolation

2. Other cryptographic assets, for example, intermediate calculation values and RNG state. Disclosure or modification of these assets can enable recovery of cryptographic keys, and loss of

cryptographic protection.

3. Application input/output data and cryptographic operation state.

   Application data is only provided to the Cryptoprocessor for cryptographic operations, and its stakeholder is the application owner.

   Disclosure of this data — whether it is plaintext, or other data or state — to an attacker defeats the protection that the use of cryptography provides. Modification of this data can have the same effect.

D.1.3 Security goals

Cryptography is used as a mitigation to the risk of disclosure or tampering with data assets that require protection, where isolation of the attacker from the data asset is unavailable or inadequate. Using cryptography introduces new threats related to the incorrect use of cryptography and mismanagement of cryptographic keys. Table 44 lists the security goals for the Crypto API to address these threats.

Table 44 Security goals

Id	Description
G.1	An attacker shall not be able to disclose the plaintext corresponding to a ciphertext for which they do not own the correct key.
G.2	An attacker shall not be able to generate authenticated material for which they do not own the correct key.
G.3	An attacker shall not be able to exfiltrate keys or other private information stored by the Crypto API.
G.4	An attacker shall not be able to alter any state held by the implementation of the Crypto API, such as internal keys or other private information (for example, certificates, signatures, etc.).

D.2 Threat Model

D.2.1 Adversarial models

The API itself has limited ability to mitigate threats. However, mitigation of some of the threats within the cryptoprocessor can place requirements on the API design. This analysis considers a broad attack surface, to also identify requirements that enable the mitigation of specific threats within a cryptoprocessor implementation.

Table 45 on page 440 describes the adversarial models that are considered in this assessment.

A specific implementation of the Crypto API might not include all of these adversarial models within its own threat model. In this case, the related threats, risks, and mitigations might not be required for that implementation.

Table 45 Adversarial models

Id	Description
M.0	The Adversary is capable of accessing data that is outside the Security Perimeter of the system and on commonly accessible channels, such as messages in transit or data in storage. This includes, but is not limited to: • Read any input and output. • Provide, forge, replay or modify input. • Attempt to gain read/write access to external storage devices. • Perform timings on the operations being done by the target machine, either in normal operation or as a response to crafted inputs. For example, timing attacks on web servers. Once access to data is obtained, we do not make a further case distinction of the Adversarial Model depending on other capabilities. For example, the ability to perform cryptanalysis on intercepted ciphertext.
M.1	The Adversary is capable of mounting attacks from software. This includes, but is not limited to: • Software exploitation. • Side channel analysis that that relies on software-exposed, built-in hardware features to perform physical unit and time measurements. • Attacks that exploit access to any memory mapped configuration, monitoring, debug register. • Software-induced glitching of resources, for example Row hammer, or crashing the CPU by running intensive tasks.
M.2	The Adversary is capable of mounting simple, passive hardware attacks. This Adversary has physical access to the hardware. This includes, but is not limited to: • Side channel analyses that require external measurement devices. For example, this can utilize leakage sources such as EM emissions, power consumption, photonic emission, or acoustic channels. • Plugging malicious hardware into an unmodified system. • Passive SoC or memory interposition.

Adversarial models that are outside the scope of this assessment are shown in Table 46 on page 441.

Table 46 Adversarial models that are outside the scope of this SRA

Id	Description
M.3	The Adversary is capable of mounting sophisticated and active physical attacks. This includes, but is not limited to: • Interposing memory and blocking, replaying, and injecting transactions, this requires a much more precise timing than passive eavesdropping. • Replacing or adding chips on the motherboard.
M.4	The Adversary is capable of performing invasive silicon microsurgery.

D.2.2 Threats and attacks

Table 47 describes threats to the Security Goals, and provides examples of corresponding attacks. This table identifies which Security goals are affected by the attacks, and which Adversarial model or models are required to execute the attack.

See *Risk assessment* on page 443 for an evaluation of the risks posed by these threats, *Mitigations* on page 444 for mitigation requirements in the API design, and *Implementation remediations* on page 447 for mitigation recommendations in the cryptoprocessor implementation.

Table 47 Threats and attacks

Threat				Attack (Examples)
Id	Description	Goals	Models	Id: Description
T.1	Use of insecure or incorrectly implemented cryptography	G.1 G.2	M.0	**A.C1**: Using a cryptographic algorithm that is not adequately secure for the application use case can permit an attacker to recover the application plaintext from attacker-accessible data. **A.C2**: Using a cryptographic algorithm that is not adequately secure for the application use case can permit an attacker to inject forged authenticated material into application data in transit or in storage. **A.C3**: Using an insecure cryptographic algorithm, or one that is incorrectly implemented can permit an attacker to recover the cryptographic key. Key recovery enables the attacker to reveal encrypted plaintexts, and inject forged authenticated data.
T.2	Misuse of cryptographic algorithms	G.1 G.2	M.0	**A.C4**: Reusing a cryptographic key with different algorithms can result in cryptanalysis attacks on the ciphertexts or signatures which enable an attacker to recover the plaintext, or the key itself.

continues on next page

Table 47 – continued from previous page

Threat				Attack (Examples)
Id	Description	Goals	Mod-els	Id: Description
T.3	Recover non-extractable key through the API	G.3	M.1	**A.C5**: The attacker uses an indirect mechanism provided by the API to extract a key that is not intended to be extractable. **A.C6**: The attacker uses a mechanism provided by the API to enable brute-force recovery of a non-extractable key. For example, *On the Security of PKCS #11* [CLULOW] describes various flaws in the design of the PKCS #11 interface standard that enable an attacker to recover secret and non-extractable keys.
T.4	Illegal inputs to the API	G.3 G.4	M.1	**A.60**: Using a pointer to memory that does not belong to the application, in an attempt to make the cryptoprocessor read or write memory that is inaccessible to the application. **A.70**: Passing out-of-range values, or incorrectly formatted data, to provoke incorrect behavior in the cryptoprocessor. **A.61**: Providing invalid buffer lengths to cause out-of-bounds read or write access within the cryptoprocessor. **A.62**: Call API functions in an invalid sequence to provoke incorrect operation of the cryptoprocessor.
T.5	Direct access to cryptoprocessor state	G.3 G.4	M.1	**A.C7**: Without a cryptoprocessor boundary, an attacker can directly access the cryptoprocessor state from an application. See Figure 34 on page 437. **A.C8**: A misconfigured cryptoprocessor boundary can allow an attacker to directly access the cryptoprocessor state from an Application.
T.6	Access and use another application's assets	G.1 G.2	M.1	**A.C9**: Without application boundaries, the cryptoprocessor provides a unified view of the application assets. All keys are accessible to all callers of the Crypto API. See Figure 36 on page 438. **A.C10**: The attacker can spoof the application identity within a caller-isolated implementation to gain access to another application's assets.
T.7	Data-dependent timing	G.1 G.3	M.1	**A.C11** Measuring the time for operations in the cryptoprocessor or the application, and using the differential in results to assist in recovery of the key or plaintext.

continues on next page

Table 47 – continued from previous page

Threat				Attack (Examples)
Id	Description	Goals	Mod-els	Id: Description
T.8	Memory manipulation	G.4	M.2	**A.19**: Corrupt application or cryptoprocessor state via a fault, causing incorrect operation of the cryptoprocessor.
			M.1	**A.59**: Modifying function parameters in memory, while the cryptoprocessor is accessing the parameter memory, to cause incorrect operation of the cryptoprocessor.
T.9	Side channels	G.1 G.3	M.2	**A.C12** Taking measurements from physical side-channels during cryptoprocessor operation, and using this data to recover keys or plaintext. For example, using power or EM measurements.
			M.1	**A.C13** Taking measurements from shared-resource side-channels during cryptoprocessor operation, and using this data to recover keys or plaintext. For example, attacks using a shared cache.

D.2.3 Risk assessment

The risk ratings in Table 48 follow a version of the risk assessment scheme in *NIST Special Publication 800-30 Revision 1: Guide for Conducting Risk Assessments* [SP800-30]. Likelihood of an attack and its impact are evaluated independently, and then they are combined to obtain the overall risk of the attack.

The risk assessment is used to prioritize the threats that require mitigation. This helps to identify the mitigations that have the highest priority for implementation. Mitigations are described in *Mitigations* on page 444 and *Implementation remediations* on page 447.

It is recommended that this assessment is repeated for a specific implementation or product, taking into consideration the Adversarial models that are within scope, and re-evaluating the impact based on the assets at risk. Table 48 repeats the association in Table 47 on page 441 between an Adversarial model and the Threats that it enables. This aids filtering of the assessment based on the models that are in scope for a specific implementation.

Table 48 Risk assessment

Adversarial Model	Threat/Attack	Likelihood	Impact [a]	Risk
M.0	T.1	High	Medium	Medium
M.0	T.2	High	Medium	Medium
M.1	T.3	Medium	High	Medium
M.1	T.4	High	Medium	Medium
M.1	T.5	High	Very high	Very high

continues on next page

Table 48 – continued from previous page

Adversarial Model	Threat/Attack	Likelihood	Impact [a]	Risk
M.1	T.6	High	High	High
M.1	T.7	Medium	Medium	Medium
M.1	T.8/A.59	Medium	Medium	Medium
M.2	T.8/A.19	Low	Medium	Low
M.2	T.9/A.C12	Low	High	Medium
M.1	T.9/A.C13	Medium	High	Medium

a. The impact of an attack is dependent on the impact of the disclosure or modification of the application data that is cryptographically protected. This is ultimately determined by the requirements and risk assessment for the product which is using the Crypto API. Table 48 on page 443 allocates the impact as follows:
   - 'Medium' if unspecified cryptoprocessor state or application data assets are affected.
   - 'High' if an application's cryptographic assets are affected.
   - 'Very High' if all cryptoprocessor assets are affected.

D.3 Mitigations

D.3.1 Objectives

The objectives in Table 49 are a high-level description of what the design must achieve in order to mitigate the threats. Detailed requirements that describe how the API or cryptoprocessor implementation can deliver the objectives are provided in *Requirements* on page 445 and *Implementation remediations* on page 447.

Table 49 Mitigation objectives

Id	Description	Threats addressed
O.1	Hide keys from the application	
	Keys are never directly manipulated by application software. Instead keys are referred to by handle, removing the need to deal with sensitive key material inside applications. This form of API is also suitable for secure elements, based on tamper-resistant hardware, that never reveal cryptographic keys.	T.1 T.2 T.3 — see *A keystore interface* on page 21. T.5 T.6 — to mitigate T.5 and T.6, the implementation must provide some form of isolation. See *Optional isolation* on page 21.
O.2	Limit key usage	
	Associate each key with a policy that limits the use of the key. The policy is defined by the application when the key is created, after which it is immutable.	T.2 T.3 — see *Key policies* on page 100.
O.3	Best-practice cryptography	

continues on next page

Table 49 – continued from previous page

Id	Description	Threats addressed
	An application developer-oriented API to achieve practical cryptography: the Crypto API offers services that are oriented towards the application of cryptographic methods like encrypt, sign, verify. This enables the implementation to focus on best-practice implementation of the cryptographic primitive, and the application developer on correct selection and use of those primitives.	T.1 T.2 T.7 T.8 — see *Ease of use* on page 22.
O.4	Algorithm agility	
	Cryptographic functions are not tied to a specific cryptographic algorithm. Primitives are designated at run-time. This simplifies updating an application to use a more secure algorithm, and makes it easier to implement dynamic selection of cryptographic algorithms within an application.	T.1 — see *Choice of algorithms* on page 22.

D.3.2 Requirements

The design of the API can mitigate, or enable a cryptoprocessor to mitigate, some of the identified attacks. Table 50 describes these mitigations. Mitigations that are delegated to the cryptoprocessor or application are described in *Implementation remediations* on page 447.

Table 50 Security requirements

Id	Description	API impact	Threats/attacks addressed
SR.1 (O.1)	Key values are not exposed by the API, except when importing or exporting a key.	The full key policy must be provided at the time a key is created. See *Key management* on page 24.	T.3/A.C5 — key values are hidden by the API.
SR.2 (O.2)	The policy for a key must be set when the key is created, and be immutable afterward.	The full key policy must be provided at the time a key is created. See `psa_key_attributes_t`.	T.3/A.C5 — once created, the key usage permissions cannot be changed to permit export. T.2/A.C4— once created, a key cannot be repurposed by changing its policy.
SR.3 (O.2)	The key policy must control the algorithms that the key can be used with, and the functions of the API that the key can be used with.	The key policy must include usage permissions, and permitted-algorithm attributes. See *Key policies* on page 100.	T.2/A.C4 — a key cannot be reused with different algorithms.

continues on next page

Table 50 – continued from previous page

Id	Description	API impact	Threats/attacks addressed
SR.4 (O.1)	Key export must be controlled by the key policy.	See `PSA_KEY_USAGE_EXPORT`.	T.3/A.C5 — a key can only be extracted from the cryptoprocessor if explicitly permitted by the key creator.
SR.5 (O.1)	The policy of a copied key must not provide rights that are not permitted by the original key policy.	See `psa_copy_key()`.	T.3/A.C5 — a copy of a key cannot be exported if the original could not be exported. T.3/A.C4 — a copy of a key cannot be used in different algorithm to the original.
SR.6 (O.3)	Unless explicitly required by the use case, the API must not define cryptographic algorithms with known security weaknesses. If possible, deprecated algorithms should not be included.	Algorithm inclusion is based on use cases. Warnings are provided for algorithms and operations with known security weaknesses, and recommendations made to use alternative algorithms.	T.1/A.C1 A.C2 A.C3
SR.7 (O.4)	The API design must make it easy to change to a different algorithm of the same type.	Cryptographic operation functions select the specific algorithm based on parameters passed at runtime. See *Key types* on page 53 and *Algorithms* on page 130.	T.1/A.C1 A.C2 A.C3
SR.8 (O.1)	Key-derivation functions that expose part of the key value, or make part of the key value easily recoverable, must not be provided in the API.		T.3/A.C6
SR.9 (O.3)	Constant values defined by the API must be designed to resist bit faults.	Key type values explicitly consider single-bit faults, see *Key type encoding* on page 421. [a] Success and error status codes differ by multiple bits, see *Status codes* on page 45. [b]	T.8/A.19 — enablement only, mitigation is delegated to the implementation.
SR.10 (O.3)	The API design must permit the implementation of operations with data-independent timing.	Provision of comparison functions for MAC, hash and key-derivation operations.	T.7/A.C11 — enablement only, mitigation is delegated to the implementation.

continues on next page

Table 50 – continued from previous page

Id	Description	API impact	Threats/attacks addressed
SR.11 (O.3)	Specify behavior for memory shared between the application and cryptoprocessor, including where multiple parameters overlap.	Standardize the result when parameters overlap, see *Overlap between parameters* on page 37.	T.8/A.59 — enablement only, mitigation is delegated to the implementation.
SR.12 (O.1) (O.2)	The API must permit the implementation to isolate the cryptoprocessor, to prevent access to keys without using the API.	No use of shared memory between application and cryptoprocessor, except as function parameters.	T.5/A.C7 — enablement only, mitigation is delegated to the implementation.
SR.13 (O.3)	The API design must permit the implementation of operations using mitigation techniques that resist side-channel attacks.	Operations that use random blinding to resist side-channel attacks, can return RNG-specific error codes. See also SR.12, which enables the cryptoprocessor to be fully isolated, and implemented within a separate security processor.	T.9 — enablement only, mitigation is delegated to the implementation.

a. Limited resistance to bit faults is still valuable in systems where memory may be susceptible to single-bit flip attacks, for example, Rowhammer on some types of DRAM.

b. Unlike key type values, algorithm identifiers used in cryptographic operations are verified against a the permitted-algorithm in the key policy. This provides a mitigation for a bit fault in an algorithm identifier value, without requiring error detection within the algorithm identifier itself.

D.4 Remediation & residual risk

D.4.1 Implementation remediations

Table 51 on page 448 includes all recommended remediations for an implementation, assuming the full adversarial model described in *Adversarial models* on page 439. When an implementation has a subset of the adversarial models, then individual remediations can be excluded from an implementation, if the associated threat is not relevant for that implementation.

Table 51 Implementation remediations

Id	Identified gap	Suggested remediation
R.1 (O.1) (O.3)	T.5 — direct access to cryptoprocessor state.	The cryptoprocessor implementation provides *cryptoprocessor isolation* or *caller isolation*, to isolate the application from the cryptoprocessor state, and from volatile and persistent key material.
R.2 (O.1) (O.3)	T.6 — access and use another application's assets.	The cryptoprocessor implementation provides *caller isolation*, and maintains separate cryptoprocessor state for each application. Each application must only be able to access its own keys and ongoing operations. Caller isolation requires that the implementation can securely identify the caller of the Crypto API.
R.3 (O.3)	T.4/A.60 A.61 — using illegal memory inputs.	The cryptoprocessor implementation validates that memory buffers provided by the application are accessible by the application.
R.4 (O.3)	T.4/A.70 — providing invalid formatted data.	The cryptoprocessor implementation checks that imported key data is valid before use.
R.5 (O.3)	T.4/A.62 — call the API in an invalid operation sequence.	The cryptoprocessor implementation enforces the correct sequencing of calls in multi-part operations. See *Multi-part operations* on page 27.
R.6 (O.1) (O.3)	T.3/A.C5 A.C6 — indirect key disclosure via the API.	Cryptoprocessor implementation-specific extensions to the API must avoid providing mechanisms that can extract or recover key values, such as trivial key-derivation algorithms.
R.8 (O.3)	T.8/A.59 — concurrent modification of parameter memory.	The cryptoprocessor implementation treats application memory as untrusted and volatile, typically by not reading the same memory location twice. See *Stability of parameters* on page 37.
R.9 (O.3)	T.2/A.C4 — incorrect cryptographic parameters.	The cryptoprocessor implementation validates the key attributes and other parameters used for a cryptographic operation, to ensure these conform to the API specification and to the specification of the algorithm itself.
R.10 (O.3)	T.1/A.C1 A.C2 A.C3 — insecure cryptographic algorithms.	The cryptoprocessor does not support deprecated cryptographic algorithms, unless justified by specific use case requirements.
R.11 (O.3)	T.7/A.C11 — data-independent timing.	The cryptoprocessor implements cryptographic operations with data-independent timing.
R.12 (O.3)	T.9 — side-channels.	The cryptoprocessor implements resistance to side-channels.

D.4.2 Residual risk

Threats T.2-T.4, and T.7-T.9 are fully mitigated in the API design, as described in *Mitigations* on page 444, or the cryptoprocessor implementation, as described in *Implementation remediations* on page 447.

Table 52 describes the remaining risks related to T.1, T.5, and T.6 that cannot be mitigated fully by the API or cryptoprocessor implementation. Responsibility for managing these risks lies with the application developers and system integrators.

Table 52 Residual risk

Id	Threat/attack	Suggested remediations
RR.1	T.1	Selection of appropriately secure protocols, algorithms and key sizes is the responsibility of the application developer.
RR.2	T.5	Correct isolation of the cryptoprocessor is the responsibility of the cryptoprocessor and system implementation.
RR.3	T.6	Correct identification of the application client is the responsibility of the cryptoprocessor and system implementation.

Appendix E: Changes to the API

E.1 Document change history

This section provides the detailed changes made between published version of the document.

E.1.1 Changes between *1.3.2* and *1.4.0*

Changes to the API

- Added `psa_attach_key()` to register existing key material as a volatile key within the implementation.
- Added `psa_check_key_usage()` to query a key's capabilities.
- Add support for extendable-output functions (XOF). See *Extendable-output functions (XOF)* on page 157.
- Added support for key wrapping using key-wrapping algorithms. See *Key wrapping* on page 237.
- Added support for context parameters in signature algorithms:
  - `psa_sign_message_with_context()`
  - `psa_verify_message_with_context()`
  - `psa_sign_hash_with_context()`
  - `psa_verify_hash_with_context()`

  See *Asymmetric signature* on page 278.
- Added PureEdDSA algorithms with non-zero context. See *EdDSA signature algorithms* on page 289 and `PSA_ALG_EDDSA_CTX`.

- Added support for the WPA3-SAE PAKE:
  - Add `PSA_KEY_TYPE_WPA3_SAE_ECC` and `PSA_KEY_TYPE_WPA3_SAE_DH` key types for WPA3-SAE password tokens.
  - Added the `PSA_ALG_WPA3_SAE_H2E()` KDF for generating a WPA3-SAE password token from a password.
  - Added WPA3-SAE PAKE algorithms, `PSA_ALG_WPA3_SAE_FIXED()` and `PSA_ALG_WPA3_SAE_GDH()`.
  - Added finite field Diffie-Hellman family `PSA_DH_FAMILY_RFC3526`, which provides cyclic groups used for WPA3-SAE.
  - Added wildcard key policy `PSA_ALG_WPA3_SAE_ANY` to permit password and password token keys to be used in any WPA3-SAE cipher suite.

  See *The WPA3-SAE protocol* on page 381.
- Add support for the Ascon family of light-weight algorithms:
  - `PSA_ALG_ASCON_AEAD128`
  - `PSA_ALG_ASCON_HASH256`
  - `PSA_ALG_ASCON_XOF128`
  - `PSA_ALG_ASCON_CXOF128`

Relaxations

- Relaxed the permitted-key policy requirements for ECDSA verification, to be consistent with those for ML-DSA and SLH-DSA. When verifying a signature, the `PSA_ALG_ECDSA` and `PSA_ALG_DETERMINISTIC_ECDSA` are considered equivalent when checking the key's permitted-algorithm policy.

Clarifications and fixes

- Corrected the example implementation of `PSA_ALG_IS_SIGN_HASH()` in *Example macro implementations* on page 427, to exclude PureEdDSA.
- Clarified the use of hash algorithms with `PSA_ALG_HMAC`.

Other changes

- Reorganised the chapter on key types. See *Key types* on page 53.

E.1.2 Changes between *1.3.1* and *1.3.2*

Other changes

- Updated introduction to reflect GlobalPlatform assuming the governance of the PSA Certified evaluation scheme.

E.1.3 Changes between *1.3.0* and *1.3.1*

Clarifications and fixes

- Clarify the way a 'volatile key' is designated, based on a persistence level of `PSA_KEY_PERSISTENCE_VOLATILE`, to ensure that this is consistent throughout the specification. See *Key lifetimes* on page 90.

- Corrected the type of the key id parameter to `psa_generate_key_custom()` and `psa_key_derivation_output_key_custom()`.
- Added missing 'Added in version' information to key derivation macros.

E.1.4 Changes between *1.2.1* and *1.3.0*

Changes to the API

- Added `PSA_EXPORT_ASYMMETRIC_KEY_MAX_SIZE` to evaluate the export buffer size for any asymmetric key pair or public key.
- Add extended key-generation and key-derivation functions, `psa_generate_key_custom()` and `psa_key_derivation_output_key_custom()`, that accept additional parameters to control the key creation process.
- Define a key production parameter to select a non-default exponent for RSA key generation.
- Reworked the allocation of bits in the encoding of asymmetric keys, to increase the scope for additional asymmetric key types:
  - Bit 7 was previously an unused indicator for IMPLEMENTATION DEFINED family values, and is now allocated to the ASYM-TYPE.
  - ASYM-TYPE 0 is now a category for non-parameterized asymmetric keys, of which RSA is one specific type.

  This has no effect on any currently allocated key type values, but affects the correct implementation of macros used to manipulate asymmetric key types.

  See *Asymmetric key encoding* on page 424 and *Key type macros* on page 433.
- Added key-encapsulation functions, `psa_encapsulate()` and `psa_decapsulate()`.
  - Added `PSA_ALG_ECIES_SEC1` as a key-encapsulation algorithm that implements the key agreement steps of ECIES.

Clarifications and fixes

- Clarified the documentation of key attributes in key creation functions.
- Clarified the constraint on `psa_key_derivation_output_key()` for algorithms that have a `PSA_KEY_DERIVATION_INPUT_PASSWORD` input step.
- Removed the redundant key input constraints on `psa_key_derivation_verify_bytes()` and `psa_key_derivation_verify_key()`. These match the policy already checked in `psa_key_derivation_input_key()`.
- Documented the use of context parameters in J-PAKE and SPAKE2+ PAKE operations. See *J-PAKE operation* on page 366 and *SPAKE2+ operation* on page 374.
- Clarified asymmetric signature support by categorizing the different types of signature algorithm.

Other changes

- Integrated the PAKE Extension with the main specification for the Crypto API.
- Moved the documentation of key formats and key-derivation procedures to sub-sections within each key type.

- Clarified the flexibility for an implementation to return either PSA_ERROR_NOT_SUPPORTED or PSA_ERROR_INVALID_ARGUMENT when provided with unsupported algorithm identifier or key parameters.
- Added API version information to APIs that have been added or changed since version 1.0 of the Crypto API.

E.1.5 Changes between *1.2.0* and *1.2.1*

Clarifications and fixes

- Fix the example implementation of PSA_ALG_KEY_AGREEMENT_GET_BASE() and PSA_ALG_KEY_AGREEMENT_GET_KDF() in *Example macro implementations* on page 427, to give correct results for key agreements combined with PBKDF2.
- Remove the dependency on the underlying hash algorithm in definition of HMAC keys, and their behavior on import and export. Transferred the responsibility for truncating over-sized HMAC keys to the application. See PSA_KEY_TYPE_HMAC.
- Rewrite the description of PSA_ALG_CTR, to clarify how to use the API to set the appropriate IV for different application use cases.

E.1.6 Changes between *1.1.2* and *1.2.0*

Changes to the API

- Added psa_key_agreement() for standalone key agreement that outputs to a new key object. Also added PSA_ALG_IS_STANDALONE_KEY_AGREEMENT() as a synonym for PSA_ALG_IS_RAW_KEY_AGREEMENT().
- Added support for the XChaCha20 cipher and XChaCha20-Poly1305 AEAD algorithms. See PSA_KEY_TYPE_XCHACHA20 and PSA_ALG_XCHACHA20_POLY1305.
- Added support for *zigbee Specification* [ZIGBEE] cryptographic algorithms. See PSA_ALG_AES_MMO_ZIGBEE and PSA_ALG_CCM_STAR_NO_TAG.
- Defined key-derivation algorithms based on the Counter mode recommendations in *NIST Special Publication 800-108r1: Recommendation for Key Derivation Using Pseudorandom Functions* [SP800-108]. See PSA_ALG_SP800_108_COUNTER_HMAC() and PSA_ALG_SP800_108_COUNTER_CMAC.
- Added support for TLS 1.2 ECJPAKE-to-PMS key-derivation. See PSA_ALG_TLS12_ECJPAKE_TO_PMS.
- Changed the policy for psa_key_derivation_verify_bytes() and psa_key_derivation_verify_key(), so that these functions are also permitted when an input key has the PSA_KEY_USAGE_DERIVE usage flag.
- Removed the special treatment of PSA_ERROR_INVALID_SIGNATURE for key-derivation operations. A verification failure in psa_key_derivation_verify_bytes() and psa_key_derivation_verify_key() now puts the operation into an error state.

Clarifications and fixes

- Clarified the behavior of a key-derivation operation when there is insufficient capacity for a call to psa_key_derivation_output_bytes(), psa_key_derivation_output_key(), psa_key_derivation_verify_bytes(), or psa_key_derivation_verify_key().
- Reserved the value 0 for most enum-like integral types.
- Changed terminology for clarification: a 'raw key agreement' algorithm is now a 'standalone key agreement', and a 'full key agreement' is a 'combined key agreement'.

E.1.7 Changes between *1.1.1* and *1.1.2*

Clarifications and fixes

- Clarified the requirements on the `hash` parameter in the `psa_sign_hash()` and `psa_verify_hash()` functions.
- Explicitly described the handling of input and output in `psa_cipher_update()`, consistent with the documentation of `psa_aead_update()`.
- Clarified the behavior of operation objects following a call to a setup function. Provided a diagram to illustrate multi-part operation states.
- Clarified the key policy requirement for `PSA_ALG_ECDSA_ANY`.
- Clarified `PSA_KEY_USAGE_EXPORT`: "it permits moving a key outside of its current security boundary". This improves understanding of why it is not only required for `psa_export_key()`, but can also be required for `psa_copy_key()` in some situations.

Other changes

- Moved the documentation of supported key import/export formats to a separate section of the specification.

E.1.8 Changes between *1.1.0* and *1.1.1*

Changes to the API

- Extended `PSA_ALG_TLS12_PSK_TO_MS` to support TLS cipher suites that mix a key exchange with a pre-shared key.
- Added a new key-derivation input step `PSA_KEY_DERIVATION_INPUT_OTHER_SECRET`.
- Added new algorithm families `PSA_ALG_HKDF_EXTRACT` and `PSA_ALG_HKDF_EXPAND` for protocols that require the two parts of HKDF separately.

Other changes

- Relicensed the document under Attribution-ShareAlike 4.0 International with a patent license derived from Apache License 2.0. See *License* on page x.
- Adopted a standard set of Adversarial models for the Security Risk Assessment. See *Adversarial models* on page 439.

E.1.9 Changes between *1.0.1* and *1.1.0*

Changes to the API

- Relaxation when a raw key agreement is used as a key's permitted-algorithm policy. This now also permits the key agreement to be combined with any key-derivation algorithm. See `PSA_ALG_FFDH` and `PSA_ALG_ECDH`.
- Provide wildcard permitted-algorithm polices for MAC and AEAD that can specify a minimum MAC or tag length. The following elements are added to the API:
  - `PSA_ALG_AT_LEAST_THIS_LENGTH_MAC()`

  - PSA_ALG_AEAD_WITH_AT_LEAST_THIS_LENGTH_TAG()
- Added support for password-hashing and key-stretching algorithms, as key-derivation operations.
  - Added key types PSA_KEY_TYPE_PASSWORD, PSA_KEY_TYPE_PASSWORD_HASH and PSA_KEY_TYPE_PEPPER, to support use of these new types of algorithm.
  - Add key-derivation input steps PSA_KEY_DERIVATION_INPUT_PASSWORD and PSA_KEY_DERIVATION_INPUT_COST.
  - Added psa_key_derivation_input_integer() to support numerical inputs to a key-derivation operation.
  - Added functions psa_key_derivation_verify_bytes() and psa_key_derivation_verify_key() to compare derivation output data within the cryptoprocessor.
  - Added usage flag PSA_KEY_USAGE_VERIFY_DERIVATION for using keys with the new verification functions.
  - Modified the description of existing key-derivation APIs to enable the use of key-derivation functionality.
- Added algorithms PSA_ALG_PBKDF2_HMAC() and PSA_ALG_PBKDF2_AES_CMAC_PRF_128 to implement the PBKDF2 password-hashing algorithm.
- Add support for twisted Edwards Elliptic curve keys, and the associated EdDSA signature algorithms. The following elements are added to the API:
  - PSA_ECC_FAMILY_TWISTED_EDWARDS
  - PSA_ALG_PURE_EDDSA
  - PSA_ALG_ED25519PH
  - PSA_ALG_ED448PH
  - PSA_ALG_SHAKE256_512
  - PSA_ALG_IS_HASH_EDDSA()
- Added an identifier for PSA_KEY_TYPE_ARIA.
- Added PSA_ALG_RSA_PSS_ANY_SALT(), which creates the same signatures as PSA_ALG_RSA_PSS(), but permits any salt length when verifying a signature. Also added the helper macros PSA_ALG_IS_RSA_PSS_ANY_SALT() and PSA_ALG_IS_RSA_PSS_STANDARD_SALT(), and extended PSA_ALG_IS_RSA_PSS() to detect both variants of the RSA-PSS algorithm.

Clarifications and fixes

- Described the use of header files and the general API conventions. See *Library conventions* on page 32.
- Added details for SHA-512/224 to the hash suspend state. See *Hash suspend state* on page 155.
- Removed ambiguities from support macros that provide buffer sizes, and improved consistency of parameter domain definition.
- Clarified the length of salt used for creating PSA_ALG_RSA_PSS() signatures, and that verification requires the same length of salt in the signature.
- Documented the use of PSA_ERROR_INVALID_ARGUMENT when the input data to an operation exceeds the limit specified by the algorithm.
- Clarified how the PSA_ALG_RSA_OAEP() algorithm uses the hash algorithm parameter.
- Fixed error in psa_key_derivation_setup() documentation: combined key-agreement and key-derivation algorithms are valid for the Crypto API.

- Added and clarified documentation for error conditions across the API.
- Clarified the distinction between `PSA_ALG_IS_HASH_AND_SIGN()` and `PSA_ALG_IS_SIGN_HASH()`.
- Clarified the behavior of `PSA_ALG_IS_HASH_AND_SIGN()` with a wildcard algorithm policy parameter.
- Documented the use of `PSA_ALG_RSA_PKCS1V15_SIGN_RAW` with the `PSA_ALG_RSA_PKCS1V15_SIGN(PSA_ALG_ANY_HASH)` wildcard policy.
- Clarified the way that `PSA_ALG_CCM` determines the value of the CCM configuration parameter *L*. Clarified that nonces generated by `psa_aead_generate_nonce()` can be shorter than the default nonce length provided by `PSA_AEAD_NONCE_LENGTH()`.

Other changes

- Add new appendix describing the encoding of algorithm identifiers and key types. See *Algorithm and key type encoding* on page 410.
- Migrated cryptographic operation summaries to the start of the appropriate operation section, and out of the *Functionality overview* on page 24.
- Included a Security Risk Assessment for the Crypto API.

E.1.10 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.
- Clarify the requirements for implementations that support concurrent execution of API calls.

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 100.
  - Moved most introductory material from the *Functionality overview* on page 24 into the relevant API sections.

E.1.11 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 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 40.
  - 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 427.
- 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 155, and supporting macros are provided for using the Crypto 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.

E.1.12 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 Certified APIs. 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.

E.1.13 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.

E.2 Planned changes for version 1.4.x

Future versions of this specification that use a 1.4.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 might be included in a version 1.2.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.

E.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:

- Integration of the PQC extension.
- Further PQC algorithms as they are standardized.
- Interruptible (incremental) operations for long-running computation in a constrained execution context.
- Import and export of additional key formats and wrapped key structures.
- 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_A

PSA_B

PSA_I

PSA_K

PSA_M

PSA_P

PSA_R

PSA_S

PSA_T

PSA_U

PSA_V

PSA_W

PSA_X