

# PSA Certified Crypto API 1.2

Document number:	IHI 0086
Release Quality:	Final
Issue Number:	1
Confidentiality:	Non-confidential
Date of Issue:	28/03/2024

Copyright  $\ensuremath{\mathbb{C}}$  2018-2024 Arm Limited and/or its affiliates

#### Abstract

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

# Contents

Abo	out this document	vii
	Release information	vii
	License	viii
	References	ix
	Terms and abbreviations	xiii
	Potential for change	xv
	<b>Conventions</b> Typographical conventions Numbers	xv xv xvi
	Feedback	xvi
1 Intr	roduction	17
1.1	About Platform Security Architecture	17
1.2	About the Crypto API	17
2 Des	sign goals	18
2.1	Suitable for constrained devices	18
2.2	A keystore interface	18
2.3	Optional isolation	18
2.4	Choice of algorithms	19
2.5	Ease of use	20
2.6 2.6.1 2.6.2 2.6.3 2.6.4 2.6.5 2.6.6 2.6.7	Example use cases Network Security (TLS) Secure Storage Network Credentials Device Pairing Secure Boot Attestation Factory Provisioning	20 20 20 20 20 20 20 20 21 21
3 Fur	nctionality overview	21
3.1	Library management	21
<b>3.2</b> 3.2.1	<b>Key management</b> Key types	21 22

3.2.2 3.2.3 3.2.4 3.2.5	Key identifiers Key lifetimes Key policies Recommendations of minimum standards for key management	22 22 23 23
<b>3.3</b> 3.3.1 3.3.2 3.3.3	<b>Symmetric cryptography</b> Single-part Functions Multi-part operations Example of the symmetric cryptography API	23 24 24 26
3.4	Asymmetric cryptography	27
3.5	Randomness and key generation	27
4 Samı	ple architectures	27
4.1	Single-partition architecture	27
4.2	Cryptographic token and single-application processor	28
4.3	Cryptoprocessor with no key storage	28
4.4	Multi-client cryptoprocessor	28
4.5	Multi-cryptoprocessor architecture	29
5 Libra	iry conventions	29
5.1	Header files	29
5.2 5.2.1 5.2.2 5.2.3 5.2.4 5.2.5 5.2.6	API conventions Identifier names Basic types Data types Constants Function-like macros Functions	30 30 30 30 31 31 31
<b>5.3</b> 5.3.1 5.3.2	<b>Error handling</b> Return status Behavior on error	32 32 32
5.4 5.4.1 5.4.2 5.4.3 5.4.4 5.4.5	Parameter conventions Pointer conventions Input buffer sizes Output buffer sizes Overlap between parameters Stability of parameters	33 33 33 34 34 34
<b>5.5</b> 5.5.1	Key types and algorithms Structure of key types and algorithms	35 35
5.6	Concurrent calls	36

6 Impl	ementation considerations	37
<b>6.1</b> 6.1.1 6.1.2 6.1.3	Implementation-specific aspects of the interface Implementation profile Implementation-specific types Implementation-specific macros	37 37 37 37
6.2 6.2.1 6.2.2 6.2.3	<b>Porting to a platform</b> Platform assumptions Platform-specific types Cryptographic hardware support	38 38 38 39
6.3 6.3.1 6.3.2 6.3.3 6.3.4 6.3.5 6.3.6	Security requirements and recommendations Error detection Indirect object references Memory cleanup Managing key material Safe outputs on error Attack resistance	39 39 39 40 40 40
<b>6.4</b> 6.4.1	Other implementation considerations Philosophy of resource management	41 41
7 Usag	ge considerations	41
<b>7.1</b> 7.1.1 7.1.2 7.1.3	Security recommendations Always check for errors Shared memory and concurrency Cleaning up after use	41 41 42 42
8 Libra	ary management reference	42
8.1 8.1.1 8.1.2	Status codes Common error codes Error codes specific to the Crypto API	42 43 44
8.2 8.2.1 8.2.2	API version Library initialization	45 45
9 Key	management reference	47
<b>9.1</b> 9.1.1	<b>Key attributes</b> Managing key attributes	47 47
9.2.1 9.2.2 9.2.3 9.2.4 9.2.5	Key types Key type encoding Key categories Symmetric keys RSA keys Elliptic Curve keys	51 51 52 53 61 62

9.2.0	Diffie Hellman keys	69
9.2.7	Attribute accessors	72
<b>9.3</b>	Key lifetimes	74
9.3.1	Volatile keys	74
9.3.2	Persistent keys	75
9.3.3	Lifetime encodings	75
9.3.4	Lifetime values	78
9.3.5	Attribute accessors	80
9.3.6	Support macros	81
<b>9.4</b>	<b>Key identifiers</b>	82
9.4.1	Key identifier type	83
9.4.2	Attribute accessors	84
<b>9.5</b>	<b>Key policies</b>	85
9.5.1	Permitted algorithms	85
9.5.2	Key usage flags	87
<b>9.6</b>	<b>Key management functions</b>	92
9.6.1	Key creation	92
9.6.2	Key destruction	98
9.6.3	Key export	100
9.6.4	Key formats	105
10 Cryp	tographic operation reference	108
<b>10.1</b>	Algorithms	108
10.1.1	Algorithm encoding	109
10.1.2	Algorithm categories	110
<b>10.1</b>	Algorithms	108
10.1.1	Algorithm encoding	109
10.1.2	Algorithm categories	110
<b>10.2</b>	Message digests (Hashes)	114
10.2.1	Hash algorithms	115
10.2.2	Single-part hashing functions	119
10.2.3	Multi-part hashing operations	121
10.2.4	Support macros	130
10.2.5	Hash suspend state	133
10.1 10.1.1 10.1.2 10.2 10.2.1 10.2.2 10.2.3 10.2.4 10.2.5 10.3 10.3.1 10.3.2 10.3.3 10.3.4	Algorithms Algorithm encoding Algorithm categories Message digests (Hashes) Hash algorithms Single-part hashing functions Multi-part hashing operations Support macros Hash suspend state Message authentication codes (MAC) MAC algorithms Single-part MAC functions Multi-part MAC operations Support macros	108 109 110 114 115 119 121 130 133 135 136 139 142 149
<ul> <li>10.1 10.1.1 10.1.2</li> <li>10.2 10.2.1 10.2.2 10.2.3 10.2.4 10.2.5</li> <li>10.3 10.3.1 10.3.2 10.3.3 10.3.4</li> <li>10.4 10.4.1 10.4.2 10.4.3 10.4.4</li> </ul>	Algorithms Algorithm encoding Algorithm categories Message digests (Hashes) Hash algorithms Single-part hashing functions Multi-part hashing operations Support macros Hash suspend state Message authentication codes (MAC) MAC algorithms Single-part MAC functions Multi-part MAC operations Support macros Unauthenticated ciphers Cipher algorithms Single-part cipher functions Multi-part cipher operations Support macros	108 109 110 114 115 119 121 130 133 135 136 139 142 149 149 151 152 159 163 172

10.5.2	Single-part AEAD functions	184
10.5.3	Multi-part AEAD operations	187
10.5.4	Support macros	203
<b>10.6</b>	<b>Key derivation</b>	209
10.6.1	Key derivation algorithms	210
10.6.2	Input step types	220
10.6.3	Key derivation functions	222
10.6.4	Support macros	238
<b>10.7</b>	Asymmetric signature	242
10.7.1	Asymmetric signature algorithms	242
10.7.2	Asymmetric signature functions	251
10.7.3	Support macros	258
<b>10.8</b>	Asymmetric encryption	264
10.8.1	Asymmetric encryption algorithms	264
10.8.2	Asymmetric encryption functions	265
10.8.3	Support macros	269
<b>10.9</b>	<b>Key agreement</b>	271
10.9.1	Key agreement algorithms	271
10.9.2	Standalone key agreement	274
10.9.3	Combining key agreement and key derivation	278
10.9.4	Support macros	280
<b>10.10</b>	Other cryptographic services	283
10.10.1	Random number generation	283
A Exan	nple header file	284
A.1	psa/crypto.h	284
B Algo	rithm and key type encoding	298
B.1 B.1.2 B.1.3 B.1.4 B.1.5 B.1.6 B.1.7 B.1.8 B.1.9 B.2 B.2.1	Algorithm identifier encoding Algorithm categories Hash algorithm encoding MAC algorithm encoding Cipher algorithm encoding AEAD algorithm encoding Key derivation algorithm encoding Asymmetric signature algorithm encoding Asymmetric encryption algorithm encoding Key agreement algorithm encoding Key type encoding Key type categories	298 299 299 300 301 302 303 304 305 305 305 306 306
B.2.2	Raw key encoding	307
B.2.3	Symmetric key encoding	307
B.2.4	Asymmetric key encoding	308

C Exa	ample macro implementations	310
C.1	Algorithm macros	311
C.2	Key type macros	315
C.3	Hash suspend state macros	316
D Sec	curity Risk Assessment	317
D.1.1 D.1.2 D.1.3	<b>Architecture</b> System definition Assets and stakeholders Security goals	317 317 319 320
D.2 D.2.1 D.2.2 D.2.3	<b>Threat Model</b> Adversarial models Threats and attacks Risk assessment	320 320 322 324
D.3 D.3.1 D.3.2	<b>Mitigations</b> Objectives Requirements	325 325 326
D.4 D.4.1 D.4.2	<b>Remediation &amp; residual risk</b> Implementation remediations Residual risk	329 329 330
E Cha	anges to the API	330
E.1 E.1.1 E.1.2 E.1.3 E.1.4 E.1.5 E.1.6 E.1.7 E.1.8 E.1.9	Document change history Changes between 1.2.0 and 1.2.1 Changes between 1.1.2 and 1.2.0 Changes between 1.1.1 and 1.1.2 Changes between 1.1.0 and 1.1.1 Changes between 1.0.1 and 1.1.0 Changes between 1.0.0 and 1.0.1 Changes between 1.0 beta 3 and 1.0.0 Changes between 1.0 beta 2 and 1.0 beta 3 Changes between 1.0 beta 1 and 1.0 beta 2	330 331 332 332 332 332 334 335 345 346
E.2	Planned changes for version 1.2.x	346
E.3	Future additions	347
Ind	ex of API elements	348

# 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

The detailed changes in each release are described in *Document change history* on page 330.

# PSA Certified Crypto API

Copyright © 2018-2024 Arm Limited and/or its affiliates. The copyright statement reflects the fact that some draft issues of this document have been released, to a limited circulation.

### License

#### **Text and illustrations**

Text and illustrations in this work are licensed under Attribution-ShareAlike 4.0 International (CC BY-SA 4.0). To view a copy of the license, visit creativecommons.org/licenses/by-sa/4.0.

**Grant of patent license**. Subject to the terms and conditions of this license (both the CC BY-SA 4.0 Public License and this Patent License), each Licensor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Licensed Material, where such license applies only to those patent claims licensable by such Licensor that are necessarily infringed by their contribution(s) alone or by combination of their contribution(s) with the Licensed Material to which such contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Licensed Material or a contribution incorporated within the License for that Licensed Material shall terminate as of the date such litigation is filed.

The Arm trademarks featured here are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Please visit arm.com/company/policies/trademarks for more information about Arm's trademarks.

#### 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

Source code samples in this work are licensed under the Apache License, Version 2.0 (the "License"); you may not use such samples except in compliance with the License. You may obtain a copy of the License at apache.org/licenses/LICENSE-2.0.

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

See the License for the specific language governing permissions and limitations under the License.

## References

This document refers to the following documents.

#### Table 2 Arm documents referenced by this document

Ref	Document Number	Title
[PSA-PAKE]	ARM AES 0058	PSA Certified Crypto API 1.2 PAKE Extension. arm-software.github.io/psa-api/crypto
[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

#### 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, <i>On the Security of PKCS #11</i> , 2003. link.springer.com/chapter/10.1007/978-3-540-45238-6_32
[CSTC0002]	Cryptography Standardization Technical Committee, <i>GM/T</i> 0002-2012: <i>SM4 block cipher algorithm</i> , 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, <i>Ed448-Goldilocks, a new elliptic curve</i> , NIST ECC Workshop, 2015. eprint.iacr.org/2015/625.pdf

Ref	Title
[FIPS180-4]	NIST, FIPS Publication 180-4: Secure Hash Standard (SHS), August 2015. doi.org/10.6028/NIST.FIPS.180-4
[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-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
[NTT-CAM]	NTT Corporation and Mitsubishi Electric Corporation, <i>Specification of Camellia – a</i> 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

#### Table 3 – continued from previous page

Ref	Title
[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
[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
[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
[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
[RFC8032]	IRTF, Edwards-Curve Digital Signature Algorithm (EdDSA), January 2017. tools.ietf.org/html/rfc8032.html

#### Table 3 – continued from previous page

Ref	Title
[RFC8439]	IRTF, ChaCha20 and Poly1305 for IETF Protocols, June 2018. tools.ietf.org/html/rfc8439.html
[RIPEMD]	Dobbertin, Bosselaers and Preneel, <i>RIPEMD-160</i> : A Strengthened Version of <i>RIPEMD</i> , 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
[SP800-30]	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
[SP800-56A]	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
[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, <i>zigbee Specification</i> , April 2017. csa-iot.org/wp- content/uploads/2022/01/docs-05-3474-22-0csg-zigbee-specification-1.pdf

#### Table 3 - continued from previous page

# 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 <i>cipher</i> 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 <i>symmetric</i> keys.
Byte	In this specification, a unit of storage comprising eight bits, also called an octet.
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 18.
Cipher	An algorithm used for encryption or decryption with a symmetric key.
Cryptoprocessor	The component that performs cryptographic operations. A cryptoprocessor might contain a <i>keystore</i> 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 18.
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 <i>hash</i> 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.

Term	Meaning
Isolation	Property of an implementation in which there is a security boundary between the application and the cryptoprocessor.
	See Optional isolation on page 18.
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
Keytune	Key metadata that describes the structure and content of a key
Keystore	A bardware or software component that protects stores and manages
Reystore	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 <i>symmetric</i> key.
Message digest	A <i>hash</i> 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.
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.
564	See Key lifetimes on page 74.
PSA	Platform Security Architecture

#### Table 4 – continued from previous page

Term	Meaning
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 <i>asymmetric</i> 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 74.

#### Table 4 – continued from previous page

## Potential for change

The contents of this specification are stable for version 1.2.

The following may change in updates to the version 1.2 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 1.2.1).
- 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 PSA Certified 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.

# 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 18.
- A high-level overview of the functionality provided by the interface. See *Functionality overview* on page 21.
- A description of typical architectures of implementations for this specification. See *Sample architectures* on page 27.
- General considerations for implementers of this specification, and for applications that use the interface defined in this specification. See *Implementation considerations* on page 37 and *Usage considerations* on page 41.
- A detailed definition of the API. See *Library management reference* on page 42, *Key management reference* on page 47, and *Cryptographic operation reference* on page 108.

PSA Certified Crypto API 1.2 PAKE Extension [PSA-PAKE] is a companion document for version 1.2 of this specification. [PSA-PAKE] defines an API for Password Authenticated Key Establishment (PAKE) algorithms. The PAKE API is now at FINAL status, and will be included in a future version of the Crypto API specification.

In future, other 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 42.

*Future additions* on page 347 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 22.
- The key identifier that the application uses to refer to the key. See *Key identifiers* on page 22.
- A lifetime that determines when the key material is destroyed, and where it is stored. See *Key lifetimes* on page 22.
- A policy that determines how the key can be used. See *Key policies* on page 23.

Keys are created using one of the key creation functions:

- psa\_import\_key()
- psa\_generate\_key()
- psa\_key\_derivation\_output\_key()
- psa\_key\_agreement()
- psa\_copy\_key()

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

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

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

## 3.2.1 Key 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 51.

## 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 82.

## 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 74.

#### 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 85.

#### 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 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 114.
- Message authentication codes (MAC). See *Message authentication codes* (MAC) on page 135.
- Symmetric ciphers. See Unauthenticated ciphers on page 151.
- Authenticated encryption with associated data (AEAD). See Authenticated encryption with associated data (AEAD) on page 178.
- Key derivation. See *Key derivation* on page 209.

For each type of symmetric cryptographic operation, the API can include:

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

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

### 3.3.1 Single-part Functions

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

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

#### 3.3.2 Multi-part operations

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

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

Each multi-part operation defines a specific object type to maintain the state of the operation. These types are implementation-defined.

All multi-part operations follow the same pattern of use, which is shown in Figure 1 on page 25.

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\_INIT macro as the type is implementation-defined.
  - Set it to all-bits zero. This is automatic if the object was allocated with calloc().
  - Assign the value of the associated macro PSA\_XXX\_INIT.
  - Assign the result of calling the associated function psa\_xxx\_init().

The resulting object is now *inactive*.

It is an error to initialize an operation object that is in *active* or *error* states. This can leak memory or other resources.

3. **Setup:** Start a new multi-part operation on an *inactive* operation object. Each operation object will define one or more setup functions to start a specific operation.

On success, a setup function will put an operation object into an *active* state. On failure, the operation object will remain *inactive*.

4. **Update:** Update an *active* operation object. The update function can provide additional parameters, supply data for processing or generate outputs.

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



Figure 1 General state model for a multi-part operation

- 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 36.
- 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 Example of the symmetric cryptography API

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

- 1. Derive the message encryption material from the master key.
  - a. Initialize a psa\_key\_derivation\_operation\_t object to zero or to PSA\_KEY\_DERIVATION\_OPERATION\_INIT.
  - b. Call psa\_key\_derivation\_setup() with PSA\_ALG\_HKDF as the algorithm.
  - c. Call psa\_key\_derivation\_input\_key() with the step PSA\_KEY\_DERIVATION\_INPUT\_SECRET and the master key.
  - d. Call psa\_key\_derivation\_input\_bytes() with the step PSA\_KEY\_DERIVATION\_INPUT\_INFO and a public value that uniquely identifies the message.
  - e. Populate a psa\_key\_attributes\_t object with the derived message encryption key's attributes.
  - f. Call psa\_key\_derivation\_output\_key() to create the derived message key.
  - g. Call psa\_key\_derivation\_output\_bytes() to generate the derived IV.
  - h. Call psa\_key\_derivation\_abort() to release the key derivation operation memory.
- 2. Encrypt the message with the derived material.
  - a. Initialize a psa\_cipher\_operation\_t object to zero or to PSA\_CIPHER\_OPERATION\_INIT.
  - b. Call psa\_cipher\_encrypt\_setup() with the derived message encryption key.
  - c. Call psa\_cipher\_set\_iv() using the derived IV retrieved above.
  - d. Call psa\_cipher\_update() one or more times to encrypt the message.
  - e. Call psa\_cipher\_finish() at the end of the message.
- 3. Call psa\_destroy\_key() to clear the generated key.

# 3.4 Asymmetric cryptography

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

- Asymmetric encryption (also known as public key encryption). See Asymmetric encryption on page 264.
- Asymmetric signature. See Asymmetric signature on page 242.
- Two-way key agreement (also known as key establishment). See Key agreement on page 271.

For asymmetric encryption and 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.

# 3.5 Randomness and key generation

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

It is recommended that applications use psa\_generate\_key(), psa\_cipher\_generate\_iv() or psa\_aead\_generate\_nonce() to generate suitably-formatted random data, as applicable. In addition, the API includes a function psa\_generate\_random() to generate and extract arbitrary random data.

# 4 Sample architectures

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

# 4.1 Single-partition architecture

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

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

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

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

# 4.2 Cryptographic token and single-application processor

This system is composed of two partitions: one is a cryptoprocessor and the other partition runs an application. There is a security boundary between the two partitions, so that the application cannot access the cryptoprocessor, except through its public interface. Thus, the architecture provides *cryptoprocessor isolation*. The cryptoprocessor has some non-volatile storage, a TRNG, and possibly, some cryptographic accelerators.

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

The 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*. 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 284 provides an incomplete, example header file which includes all of the API elements. See also *Implementation considerations* on page 37.

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 42.

# 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 37.

### 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* on page 32.

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 33.

# 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.

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 24.
- 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 34.

#### 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 298 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 310.

• 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* on page 39) 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 34.

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 36.

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 34.

# 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 44.

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)
```

(continues on next page)

#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].

Error code	Meaning in the Crypto API
PSA_ERROR_INVALID_HANDLE	A key identifier does not refer to an existing key. See also <i>Key identifiers</i> on page 22.
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 <pre>psa_crypto_init()</pre> .
PSA_ERROR_BUFFER_TOO_SMALL	Applications can call the PSA_xxx_SIZE macro listed in the function description to determine a sufficient buffer size.

continues on next page

Table 5 – continued from previous page				
Error code	Meaning in the Crypto API			
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 2

# 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 51
- Key identifiers on page 82
- Key lifetimes on page 74
- Key policies on page 85

# 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	
usage flags	$\emptyset$ — 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. Set the key policy with psa\_set\_key\_usage\_flags() and psa\_set\_key\_algorithm().
- 4. Set the key type with psa\_set\_key\_type(). Skip this step if copying an existing key with psa\_copy\_key().
- 5. When generating a random key with psa\_generate\_key() or deriving a key with psa\_key\_derivation\_output\_key(), set the desired key size with psa\_set\_key\_bits().
- 6. Call a key creation function: psa\_import\_key(), psa\_generate\_key(), psa\_key\_derivation\_output\_key(), psa\_key\_agreement(), or psa\_copy\_key(). This function reads the attribute object, creates a key with these attributes, and outputs an identifier for the newly created key.
- 7. Optionally call psa\_reset\_key\_attributes(), now that the attribute object is no longer needed. Currently this call is not required as the attributes defined in this specification do not require additional resources beyond the object itself.

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

- 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.

#### 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.
<b>Returns:</b> 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 <pre>psa_crypto_init().</pre>
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 bitfield that identifies the category and type of key. The range of key type values is divided as follows:

```
PSA_KEY_TYPE_NONE == 0
```

Reserved as an invalid key type.

0x0001 - 0x7fff

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

0x8000 - 0xffff

Implementation-defined key types. Implementations that define additional key types must use an encoding with bit 15 set. The related support macros will be easier to write if these key encodings also respect the bitwise structure used by standard encodings. The Algorithm and key type encoding on page 298 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

#### 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 Symmetric keys on page 53 for a list of symmetric key types.

#### PSA\_KEY\_TYPE\_IS\_ASYMMETRIC (macro)

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

#define PSA\_KEY\_TYPE\_IS\_ASYMMETRIC(type) /\* specification-defined value \*/

#### Parameters

type

A key type: a value of type psa\_key\_type\_t.

# Description

See RSA keys on page 61 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 Symmetric keys

#### PSA\_KEY\_TYPE\_RAW\_DATA (macro)

Raw data.

#define PSA\_KEY\_TYPE\_RAW\_DATA ((psa\_key\_type\_t)0x1001)

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

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

#### **Compatible algorithms**

PSA\_ALG\_HKDF (non-secret inputs) PSA\_ALG\_HKDF\_EXPAND (non-secret inputs) PSA\_ALG\_HKDF\_EXTRACT (non-secret inputs) PSA\_ALG\_SP800\_108\_COUNTER\_HMAC (non-secret inputs) PSA\_ALG\_SP800\_108\_COUNTER\_CMAC (non-secret inputs) PSA\_ALG\_TLS12\_PRF (non-secret inputs) PSA\_ALG\_TLS12\_PSK\_TO\_MS (non-secret inputs)

#### 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)

#### 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**

PSA\_ALG\_HKDF (secret input) PSA\_ALG\_HKDF\_EXPAND (secret input) PSA\_ALG\_HKDF\_EXTRACT (secret input) PSA\_ALG\_TLS12\_PRF (secret input) PSA\_ALG\_TLS12\_PSK\_TO\_MS (secret input)

### PSA\_KEY\_TYPE\_PASSWORD (macro)

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

```
#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**

PSA\_ALG\_PBKDF2\_HMAC() (password input) PSA\_ALG\_PBKDF2\_AES\_CMAC\_PRF\_128 (password input)

### PSA\_KEY\_TYPE\_PASSWORD\_HASH (macro)

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

#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**

PSA\_ALG\_PBKDF2\_HMAC() (key output and verification) PSA\_ALG\_PBKDF2\_AES\_CMAC\_PRF\_128 (key output and verification)

# PSA\_KEY\_TYPE\_PEPPER (macro)

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

#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**

PSA\_ALG\_PBKDF2\_HMAC() (salt input)
PSA\_ALG\_PBKDF2\_AES\_CMAC\_PRF\_128 (salt input)

#### 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\_CBC\_NO\_PADDING PSA\_ALG\_CBC\_PKCS7 PSA\_ALG\_ECB\_NO\_PADDING PSA\_ALG\_CCM PSA\_ALG\_CCM PSA\_ALG\_GCM

### PSA\_KEY\_TYPE\_ARIA (macro)

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

#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\_ECCM PSA\_ALG\_CCM PSA\_ALG\_GCM

#### 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

### 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\_CBC\_NO\_PADDING PSA\_ALG\_CBC\_PKCS7 PSA\_ALG\_ECB\_NO\_PADDING PSA\_ALG\_ECM PSA\_ALG\_CCM PSA\_ALG\_GCM

#### 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\_CBC\_NO\_PADDING PSA\_ALG\_CBC\_PKCS7 PSA\_ALG\_ECB\_NO\_PADDING PSA\_ALG\_CCM PSA\_ALG\_CCM PSA\_ALG\_GCM

#### 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

#### 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

#### PSA\_KEY\_TYPE\_XCHACHA20 (macro)

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

#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

# 9.2.4 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

#### 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)

### 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.

# 9.2.5 Elliptic Curve keys

#### psa\_ecc\_family\_t (typedef)

The type of identifiers of an elliptic curve family.

typedef uint8\_t psa\_ecc\_family\_t;

The curve identifier is required to create an ECC key using the PSA\_KEY\_TYPE\_ECC\_KEY\_PAIR() or PSA\_KEY\_TYPE\_ECC\_PUBLIC\_KEY() macros.

The specific ECC curve within a family is identified by the key\_bits attribute of the key.

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

0x00 Reserved. Not allocated to an ECC family.

- 0x01 0x7f ECC family identifiers defined by this standard. Unallocated values in this range are reserved for future use.
- 0x80 0xff Implementations that define additional families must use an encoding in this range.

#### PSA\_KEY\_TYPE\_ECC\_KEY\_PAIR (macro)

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

#define PSA\_KEY\_TYPE\_ECC\_KEY\_PAIR(curve) /\* specification-defined value \*/

#### Parameters

curve

A value of type psa\_ecc\_family\_t that identifies the ECC curve family to be used.

#### 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**

Elliptic curve key pairs can be used in Asymmetric signature and Key agreement algorithms.

The set of compatible algorithms depends on the Elliptic curve key family. See the Elliptic curve family for details.

### 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**

Elliptic curve public keys can be used for verification in Asymmetric signature algorithms.

The set of compatible algorithms depends on the Elliptic curve key family. See each Elliptic curve family for details.

#### 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].

#### **Compatible algorithms**

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

#### 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].

#### **Compatible algorithms**

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

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].

#### **Compatible algorithms**

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

#### 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.

#### **Compatible algorithms**

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

# 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.

#### Compatible algorithms

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

### 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.

#### Compatible algorithms

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

#### 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.

# **Compatible algorithms**

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

# 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].

# Compatible algorithms

PSA\_ALG\_DETERMINISTIC\_ECDSA PSA\_ALG\_ECDSA PSA\_ALG\_ECDSA\_ANY PSA\_ALG\_ECDH (key pair only)

# 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].

#### Compatible algorithms

PSA\_ALG\_ECDH (key pair only)

### PSA\_ECC\_FAMILY\_TWISTED\_EDWARDS (macro)

Twisted Edwards curves.

#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].

#### Compatible algorithms

PSA\_ALG\_PURE\_EDDSA PSA\_ALG\_ED25519PH (Edwards25519 only) PSA\_ALG\_ED448PH (Edwards448 only)

#### 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.

# 9.2.6 Diffie Hellman keys

#### 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 finite-field Diffie-Hellman key using the PSA\_KEY\_TYPE\_DH\_KEY\_PAIR() or PSA\_KEY\_TYPE\_DH\_PUBLIC\_KEY() macros.

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

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

- 0x00 Reserved. Not allocated to a DH group family.
- 0x01 0x7f DH group family identifiers defined by this standard. Unallocated values in this range are reserved for future use.

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

#### PSA\_KEY\_TYPE\_DH\_KEY\_PAIR (macro)

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

#define PSA\_KEY\_TYPE\_DH\_KEY\_PAIR(group) /\* specification-defined value \*/

#### Parameters

group

A value of type psa\_dh\_family\_t that identifies the Diffie-Hellman group family to be used.

#### Compatible algorithms

PSA\_ALG\_FFDH

#### PSA\_KEY\_TYPE\_DH\_PUBLIC\_KEY (macro)

Finite-field Diffie-Hellman public key.

#define PSA\_KEY\_TYPE\_DH\_PUBLIC\_KEY(group) /\* specification-defined value \*/

#### Parameters

group

A value of type psa\_dh\_family\_t that identifies the Diffie-Hellman group family to be used.

#### **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.

#### PSA\_DH\_FAMILY\_RFC7919 (macro)

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

#define PSA\_DH\_FAMILY\_RFC7919 ((psa\_dh\_family\_t) 0x03)

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

Keys is this group can only be used with the PSA\_ALG\_FFDH key agreement algorithm.

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

#### PSA\_KEY\_TYPE\_KEY\_PAIR\_OF\_PUBLIC\_KEY (macro)

The key pair type corresponding to a public key type.

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

#### Parameters

type

A public key type or key pair type.

#### Returns

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

#### Description

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

#### PSA\_KEY\_TYPE\_PUBLIC\_KEY\_OF\_KEY\_PAIR (macro)

The public key type corresponding to a key pair type.

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

#### Parameters

type

A public key type or key pair type.

#### Returns

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

#### Description

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

#### PSA\_KEY\_TYPE\_IS\_DH (macro)

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

#define PSA\_KEY\_TYPE\_IS\_DH(type) /\* specification-defined value \*/

#### Parameters

type

A key type: a value of type psa\_key\_type\_t.

#### PSA\_KEY\_TYPE\_IS\_DH\_KEY\_PAIR (macro)

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

#define PSA\_KEY\_TYPE\_IS\_DH\_KEY\_PAIR(type) /\* specification-defined value \*/

#### Parameters

type

A key type: a value of type psa\_key\_type\_t.
### PSA\_KEY\_TYPE\_IS\_DH\_PUBLIC\_KEY (macro)

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

#define PSA\_KEY\_TYPE\_IS\_DH\_PUBLIC\_KEY(type) /\* specification-defined value \*/

#### Parameters

type

A key type: a value of type psa\_key\_type\_t.

#### PSA\_KEY\_TYPE\_DH\_GET\_FAMILY (macro)

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

#define PSA\_KEY\_TYPE\_DH\_GET\_FAMILY(type) /\* specification-defined value \*/

#### Parameters

typeA Diffie-Hellman key type: a value of type psa\_key\_type\_t such thatPSA\_KEY\_TYPE\_IS\_DH(type) is true.

#### **Returns:** psa\_dh\_family\_t

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

### 9.2.7 Attribute accessors

#### psa\_set\_key\_type (function)

Declare the type of a key.

#### 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.

### 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 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 <a href="mailto:psa\_key\_location\_t">psa\_key\_location\_t</a> for more information.

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

### 9.3.1 Volatile keys

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

Conceptually, a volatile key is stored in RAM. Volatile keys have the lifetime PSA\_KEY\_LIFETIME\_VOLATILE.

To create a volatile key:

- 1. Populate a psa\_key\_attributes\_t object with the required type, size, policy and other key attributes.
- 2. Create the key with one of the key creation functions. If successful, these functions output a transient key identifier.

To destroy a volatile key: call psa\_destroy\_key() with the key identifier. There must be a matching call to psa\_destroy\_key() for each successful call to a create a volatile key.

# 9.3.2 Persistent keys

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

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

The lifetime attribute of a persistent key indicates how and where it is stored. The default lifetime value for a persistent key is PSA\_KEY\_LIFETIME\_PERSISTENT, which corresponds to a default storage area. This specification defines how implementations can provide other lifetime values corresponding to different storage areas with different retention policies, or to secure elements with different security characteristics.

To create a persistent key:

- 1. Populate a psa\_key\_attributes\_t object with the key's type, size, policy and other attributes.
- 2. In the attributes object, set the desired lifetime and persistent identifier for the key.
- 3. Create the key with one of the key creation functions. If successful, these functions output the key identifier that was specified by the application in step 2.

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

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

By default, persistent key material is removed from volatile memory when not in use. Frequently used persistent keys can benefit from caching, depending on the implementation and the application. Caching can be enabled by creating the key with the PSA\_KEY\_USAGE\_CACHE policy. Cached keys can be removed from volatile memory by calling psa\_purge\_key(). See also *Memory cleanup* on page 39 and *Managing key material* on page 40.

# 9.3.3 Lifetime encodings

### 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 <u>psa\_key\_persistence\_t</u> 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 6.

Table 6 Key persistence level values

Persistence level	Definition
0 = PSA_KEY_PERSISTENCE_VOLATILE	Volatile key.
	A volatile key is automatically destroyed by the implementation when the application instance terminates. In particular, a volatile key is automatically destroyed on a power reset of the device.
1 = PSA_KEY_PERSISTENCE_DEFAULT	Persistent key with a default lifetime.
	Implementations should support this value if they support persistent keys at all. Applications should use this value if they have no specific needs that are only met by implementation-specific features.

Persistence level	Definition
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.

### Table 6 – continued from previous page

### 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 7 on page 78.

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.

# 9.3.4 Lifetime values

### PSA\_KEY\_LIFETIME\_VOLATILE (macro)

The default lifetime for volatile keys.

#define PSA\_KEY\_LIFETIME\_VOLATILE ((psa\_key\_lifetime\_t) 0x0000000)

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.

# 9.3.5 Attribute accessors

### psa\_set\_key\_lifetime (function)

Set the location of a persistent key.

#### Parameters

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

### Returns: void

### Description

To make a key persistent, give it a persistent key identifier by using psa\_set\_key\_id(). By default, a key that has a persistent identifier is stored in the default storage area identifier by PSA\_KEY\_LIFETIME\_PERSISTENT. Call this function to choose a storage area, or to explicitly declare the key as volatile.

This function does not access storage, it merely stores the given value in the attribute object. The persistent key will be written to storage when the attribute object is passed to a key creation function such as psa\_import\_key(), psa\_generate\_key(), psa\_key\_derivation\_output\_key(), psa\_key\_agreement(), 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.

### 9.3.6 Support macros

#### PSA\_KEY\_LIFETIME\_GET\_PERSISTENCE (macro)

Extract the persistence level from a key lifetime.

```
#define PSA_KEY_LIFETIME_GET_PERSISTENCE(lifetime) \
    ((psa_key_persistence_t) ((lifetime) & 0x00000ff)))
```

#### 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))</pre>
```

#### Parameters

persistence	The persistence level: a value of type <pre>psa_key_persistence_t.</pre>
location	The location indicator: a value of type <pre>psa_key_location_t.</pre>

#### Returns

The constructed lifetime value.

# 9.4 Key identifiers

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

```
PSA_KEY_ID_NULL = 0
```

Reserved as an invalid key identifier.

### PSA\_KEY\_ID\_USER\_MIN - PSA\_KEY\_ID\_USER\_MAX

Applications can freely choose persistent key identifiers in this range.

### PSA\_KEY\_ID\_VENDOR\_MIN - PSA\_KEY\_ID\_VENDOR\_MAX

Implementations can define additional persistent key identifiers in this range, and must allocate any volatile key identifiers from this range.

Key identifiers outside these ranges are reserved for future use.

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

If an invalid key identifier is provided as a parameter in any function, the function will return PSA\_ERROR\_INVALID\_HANDLE; except for the special case of calling psa\_destroy\_key(PSA\_KEY\_ID\_NULL), which has no effect and always returns PSA\_SUCCESS.

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

# 9.4.1 Key identifier type

### psa\_key\_id\_t (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 82.

### 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)0x0000001)

### 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)0x3ffffff)

### PSA\_KEY\_ID\_VENDOR\_MIN (macro)

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

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

### PSA\_KEY\_ID\_VENDOR\_MAX (macro)

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

#define PSA\_KEY\_ID\_VENDOR\_MAX ((psa\_key\_id\_t)0x7fffffff)

### 9.4.2 Attribute accessors

### psa\_set\_key\_id (function)

Declare a key as persistent and set its key identifier.

#### Parameters

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

Returns: void

### Description

The application must choose a value for id between PSA\_KEY\_ID\_USER\_MIN and PSA\_KEY\_ID\_USER\_MAX.

If the attribute object currently declares the key as volatile, which is the default lifetime of an attribute object, this function sets the lifetime attribute to PSA\_KEY\_LIFETIME\_PERSISTENT.

This function does not access storage, it merely stores the given value in the attribute object. The persistent key will be written to storage when the attribute object is passed to a key creation function such as psa\_import\_key(), psa\_generate\_key(), psa\_key\_derivation\_output\_key() or psa\_copy\_key().

### Implementation note

This is a simple accessor function that is not required to validate its inputs. 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.

# 9.5 Key policies

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

- A set of usage flags. See Key usage flags on page 87.
- A specific algorithm that is permitted with the key. See *Permitted algorithms*.

The policy is part of the key attributes that are managed by a psa\_key\_attributes\_t object.

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

# 9.5.1 Permitted algorithms

The permitted algorithm is encoded using a algorithm identifier, as described in Algorithms on page 108.

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

The following algorithm policies are supported:

- PSA\_ALG\_NONE does not permit any cryptographic operation with the key. The key can still be used for non-cryptographic actions such as exporting, if permitted by the usage flags.
- A specific algorithm value permits exactly that particular algorithm.
- 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.

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

### psa\_set\_key\_algorithm (function)

Declare the permitted-algorithm policy for a key.

### Parameters

attributesThe attribute object to write to.algThe 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.

# 9.5.2 Key usage flags

The usage flags are encoded in a bitmask, which has the type psa\_key\_usage\_t. Four kinds of usage flag can be specified:

- The extractable flag PSA\_KEY\_USAGE\_EXPORT determines whether the key material can be extracted 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 40.
- The other usage flags, for example, PSA\_KEY\_USAGE\_ENCRYPT and PSA\_KEY\_USAGE\_SIGN\_MESSAGE, determine whether the corresponding operation is permitted on the key.

### psa\_key\_usage\_t (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 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)0x0000002)

This 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 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 40.

### PSA\_KEY\_USAGE\_ENCRYPT (macro)

Permission to encrypt a message with the key.

#define PSA\_KEY\_USAGE\_ENCRYPT ((psa\_key\_usage\_t)0x00000100)

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

For a key pair, this concerns the public key.

### PSA\_KEY\_USAGE\_DECRYPT (macro)

Permission to decrypt a message with the key.

#define PSA\_KEY\_USAGE\_DECRYPT ((psa\_key\_usage\_t)0x00000200)

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

For a key pair, this concerns the private key.

### PSA\_KEY\_USAGE\_SIGN\_MESSAGE (macro)

Permission to sign a message with the key.

#define PSA\_KEY\_USAGE\_SIGN\_MESSAGE ((psa\_key\_usage\_t)0x00000400)

This flag 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 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 flag is required to use the key to sign a message hash in an asymmetric signature operation. The flag must be present on keys used when calling psa\_sign\_hash().

This flag automatically sets PSA\_KEY\_USAGE\_SIGN\_MESSAGE: if an application sets the flag PSA\_KEY\_USAGE\_SIGN\_HASH when creating a key, then the key always has the permissions conveyed by PSA\_KEY\_USAGE\_SIGN\_MESSAGE, and the flag PSA\_KEY\_USAGE\_SIGN\_MESSAGE will also be present when the application queries the usage flags of the key.

For a key pair, this concerns the private key.

### PSA\_KEY\_USAGE\_VERIFY\_HASH (macro)

Permission to verify a message hash with the key.

#define PSA\_KEY\_USAGE\_VERIFY\_HASH ((psa\_key\_usage\_t)0x00002000)

This flag is required to use the key to verify a message hash in an asymmetric signature verification operation. The flag must be present on keys used when calling psa\_verify\_hash().

This flag automatically sets PSA\_KEY\_USAGE\_VERIFY\_MESSAGE: if an application sets the flag PSA\_KEY\_USAGE\_VERIFY\_HASH when creating a key, then the key always has the permissions conveyed by PSA\_KEY\_USAGE\_VERIFY\_MESSAGE, and the flag PSA\_KEY\_USAGE\_VERIFY\_MESSAGE will also be present when the application queries the usage flags of the key.

For a key pair, this concerns the public key.

### PSA\_KEY\_USAGE\_DERIVE (macro)

Permission to derive other keys or produce a password hash from this key.

#define PSA\_KEY\_USAGE\_DERIVE ((psa\_key\_usage\_t)0x00004000)

This 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\_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.

#define PSA\_KEY\_USAGE\_VERIFY\_DERIVATION ((psa\_key\_usage\_t)0x00008000)

This 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\_set\_key\_usage\_flags (function)

Declare usage flags for a key.

#### 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 85.

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.

# 9.6 Key management functions

### 9.6.1 Key creation

New keys can be created in the following ways:

- psa\_import\_key() creates a key from a data buffer provided by the application.
- psa\_generate\_key() creates a key from randomly generated data.
- psa\_key\_derivation\_output\_key() creates a key from data generated by a pseudorandom derivation process. See *Key derivation* on page 209.
- psa\_key\_agreement() creates a key from the shared secret result of a key agreement process. See *Key* agreement on page 271.
- psa\_copy\_key() duplicates an existing key with a different lifetime or with a more restrictive usage policy.

When creating a key, the attributes for the new key are specified in a psa\_key\_attributes\_t object. Each key creation function defines how it uses the attributes.

#### Note:

The attributes for a key are immutable after the key has been created.

The application must set the key algorithm policy and the appropriate key usage flags in the attributes in order for the key to be used in any cryptographic operations.

#### psa\_import\_key (function)

Import a key in binary format.

#### Parameters

attributes

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

- The key type is required, and determines how the data buffer is interpreted.
- The key size is always determined from the data buffer. If the key size in attributes is nonzero, it must be equal to the size determined from data.
- The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see *Permitted algorithms* on page 85.
- The key usage flags define what operations are permitted with the key, see *Key usage flags* on page 87.
- The key lifetime and identifier are required for a persistent key.

#### Note:

This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling psa\_get\_key\_attributes() with the key's identifier.

data

Buffer containing the key data. The content of this buffer is interpreted according to the type declared in attributes. All implementations must support at least the format described in *Key formats* on page 105 for the chosen type. Implementations can support other formats, but be conservative in interpreting the key data: it is recommended that implementations reject content if it might be erroneous, for example, if it is the wrong type or is truncated.

data_length	Size of the data buffer in bytes.
key	On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.
Returns: psa_status_t	
PSA_SUCCESS	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_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:
	<ul> <li>The key type is invalid.</li> <li>The key size is nonzero, and is incompatible with the key data in data.</li> <li>The key lifetime is invalid.</li> <li>The key identifier is not valid for the key lifetime.</li> <li>The key usage flags include invalid values.</li> <li>The key's permitted-usage algorithm is invalid.</li> <li>The key attributes, as a whole, are invalid.</li> <li>The key data is not correctly formatted for the key type.</li> </ul>
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 supports any output from psa\_export\_key(). Refer to *Key formats* on page 105 for the format of keys.

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

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

This specification defines a single format for each key type. Implementations can optionally support other formats in addition to the standard format. It is recommended that implementations that support other

formats ensure that the formats are clearly unambiguous, to minimize the risk that an invalid input is accidentally interpreted according to a different format.

#### Note:

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

#### psa\_generate\_key (function)

Generate a key or key pair.

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

c 11

#### **Parameters**

attributes

	follows:
	<ul> <li>The key type is required. It cannot be an asymmetric public key.</li> <li>The key size is required. It must be a valid size for the key type.</li> <li>The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see <i>Permitted algorithms</i> on page 85.</li> <li>The key usage flags define what operations are permitted with the key, see <i>Key usage flags</i> on page 87.</li> <li>The key lifetime and identifier are required for a persistent key.</li> </ul>
	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 <pre>psa_get_key_attributes()</pre> 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 library requires initializing by a call to <pre>psa_crypto_init().</pre>
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

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

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 following type-specific considerations apply:

• For RSA keys (PSA\_KEY\_TYPE\_RSA\_KEY\_PAIR), the public exponent is 65537. The modulus is a product of two probabilistic primes between  $2^{n-1}$  and  $2^n$  where n is the bit size specified in the attributes.

### psa\_copy\_key (function)

Make a copy of a key.

<pre>psa_status_t psa_cop</pre>	<pre>y_key(psa_key_id_t source_key,</pre>
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. This function uses the attributes as follows:
	• The key type and size can be 0. If either is nonzero, it must match the corresponding attribute of the source key.

	<ul> <li>The key location (the lifetime and, for persistent keys, the key identifier) is used directly.</li> </ul>	
	• The key policy (usage flags and permitted algorithm) are combined from the source key and attributes so that both sets of restrictions apply, as described in the documentation of this function.	
	<b>Note:</b> 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 <pre>psa_crypto_init().</pre>	
PSA_ERROR_INVALID_HANDLE	source_key is not a valid key identifier.	
PSA_ERROR_NOT_PERMITTED	The following conditions can result in this error:	
	<ul> <li>source_key does not have the PSA_KEY_USAGE_COPY usage flag.</li> <li>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.</li> <li>The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.</li> </ul>	
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:	
	<ul> <li>attributes specifies a key type or key size which does not match the attributes of source key.</li> <li>The lifetime or identifier in attributes are invalid.</li> <li>The key policies from source_key and those specified in attributes are incompatible.</li> </ul>	
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:	
	<ul> <li>The source key storage location does not support copying to the target key's storage location.</li> <li>The key attributes, as a whole, are not supported in the target key's storage location.</li> </ul>	
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.

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.

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

# 9.6.2 Key destruction

### psa\_destroy\_key (function)

Destroy a key.

psa\_status\_t psa\_destroy\_key(psa\_key\_id\_t key);

#### Parameters

	key	Identifier of the key to erase. If this is PSA_KEY_ID_NULL, do nothing and return PSA_SUCCESS.
Re	eturns: psa_status_t	
	PSA_SUCCESS	Success. If key was a valid key identifier, then the key material that it referred to has been erased. Alternatively, key was PSA_KEY_ID_NULL.
	PSA_ERROR_BAD_STATE	The library requires initializing by a call to <pre>psa_crypto_init().</pre>
	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

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.
Re	<b>eturns:</b> 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 <pre>psa_crypto_init().</pre>
	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 40.

### 9.6.3 Key export

#### psa\_export\_key (function)

Export a key in binary format.

#### 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:
	<ul> <li>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.</li> <li>PSA_EXPORT_KEY_PAIR_MAX_SIZE evaluates to the maximum output size of any supported key pair.</li> </ul>
	<ul> <li>PSA_EXPORT_PUBLIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key.</li> <li>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.</li> </ul>
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 <pre>psa_crypto_init().</pre>
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() or PSA_EXPORT_KEY_PAIR_MAX_SIZE can be used to determine a sufficient buffer size.
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>The key's storage location does not support export of the key.</li> <li>The implementation does not support export of keys with this key type.</li> </ul>
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 formats* on page 105, not the originally imported representation.

For standard key types, the output format is defined in *Key formats* on page 105.

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.

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:
	<ul> <li>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. </li> <li>PSA_EXPORT_PUBLIC_KEY_MAX_SIZE evaluates to the maximum output size of any supported public key or public part of a key pair.</li> </ul>
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() or PSA_EXPORT_PUBLIC_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:
	<ul><li>The key's storage location does not support export of the key.</li><li>The implementation does not support export of keys with this</li></ul>

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 formats* on page 105, not the originally imported representation.

For standard key types, the output format is defined in Key formats on page 105.

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);
```

(continued from previous page)

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

See also PSA\_EXPORT\_KEY\_PAIR\_MAX\_SIZE and PSA\_EXPORT\_PUBLIC\_KEY\_MAX\_SIZE.

#### PSA\_EXPORT\_PUBLIC\_KEY\_OUTPUT\_SIZE (macro)

Sufficient output buffer size for psa\_export\_public\_key().

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

#### Parameters

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

#### Returns

If the parameters are valid and supported, return a buffer size in bytes that guarantees that psa\_export\_public\_key() will not fail with PSA\_ERROR\_BUFFER\_TOO\_SMALL. If the parameters are a valid combination that is not supported by the implementation, this macro must return either a sensible size or 0. If the parameters are not valid, the return value is unspecified.

If the parameters are valid and supported, it is recommended that this macro returns the same result as PSA\_EXPORT\_KEY\_OUTPUT\_SIZE(PSA\_KEY\_TYPE\_PUBLIC\_KEY\_OF\_KEY\_PAIR(key\_type), key\_bits).

#### Description

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

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

```
if (buffer == NULL)
    handle_error(...);
size_t buffer_length;
status = psa_export_public_key(key, buffer, buffer_size, &buffer_length);
if (status != PSA_SUCCESS)
    handle_error(...);
```

See also PSA\_EXPORT\_PUBLIC\_KEY\_MAX\_SIZE.

### PSA\_EXPORT\_KEY\_PAIR\_MAX\_SIZE (macro)

Sufficient buffer size for exporting any asymmetric key pair.

#define PSA\_EXPORT\_KEY\_PAIR\_MAX\_SIZE /\* implementation-defined value \*/

This value must be a sufficient buffer size when calling psa\_export\_key() to export any asymmetric key pair that is supported by the implementation, regardless of the exact key type and key size.

See also PSA\_EXPORT\_KEY\_OUTPUT\_SIZE().

### PSA\_EXPORT\_PUBLIC\_KEY\_MAX\_SIZE (macro)

Sufficient buffer size for exporting any asymmetric public key.

#define PSA\_EXPORT\_PUBLIC\_KEY\_MAX\_SIZE /\* implementation-defined value \*/

This 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().

# 9.6.4 Key formats

This section defines the format of the key data that an implementation is required to support when importing and exporting keys. Keys can be imported using psa\_import\_key(), and exported using psa\_export\_key() or psa\_export\_public\_key(). The public key formats are also used for the key agreement functions, see *Key agreement* on page 271.

Table 8 Standard key formats

Key type	Key type details and format	
DES	PSA_KEY_TYPE_DES, 64 bits.	
	The key data consists of 8 bytes. The parity bits must be correct.	

Key type	Key type details and format
2-key 3DES	PSA_KEY_TYPE_DES, 128 bits.
3-key 3DES	PSA_KEY_TYPE_DES, 192 bits.
	The key data is the concatenation of the two or three DES keys.
Other symmetric keys	PSA_KEY_TYPE_AES
• AES	PSA_KEY_TYPE_ARC4
• ARC4	PSA_KEY_TYPE_ARIA
• ARIA	PSA_KEY_TYPE_CAMELLIA
CAMELLIA	PSA_KEY_TYPE_CHACHA20
ChaCha20	PSA_KEY_TYPE_HMAC
• HMAC	PSA_KEY_TYPE_SM4
• SM4	PSA_KEY_TYPE_DERIVE
<ul> <li>Secrets for derivation</li> </ul>	PSA_KEY_TYPE_PASSWORD_HASH
<ul> <li>Password hashes</li> </ul>	The key data is the raw bytes of the key.

Table 8 – continued from previous page

RSA key pair

### PSA\_KEY\_TYPE\_RSA\_KEY\_PAIR

The key data 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.

Key type	Key type details and format
RSA public key	PSA_KEY_TYPE_RSA_PUBLIC_KEY
	The key data 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
Weierstrass Elliptic curve key pair	PSA_KEY_TYPE_ECC_KEY_PAIR(ecc_family), where ecc_family designates a Weierstrass curve family.
	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.
Weierstrass Elliptic curve public key	PSA_KEY_TYPE_ECC_PUBLIC_KEY(ecc_family), where ecc_family designates a Weierstrass curve family.
	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 Elliptic curve	PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_MONTGOMERY)
key pair	The key data is the scalar value of the 'private key' in little-endian order as defined by <i>Elliptic Curves for Security</i> [RFC7748] §6. The value must have the forced bits set to zero or one as specified by decodeScalar25519() and decodeScalar448() in [REC7748] §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.

#### Table 8 – continued from previous page
Key type	Key type details and format
Montgomery Elliptic curve	PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_MONTGOMERY)
public key	The key data is the scalar value of the 'public key' in little-endian order as defined by <i>Elliptic Curves for Security</i> [RFC7748] §6. This is a $\lceil m/8 \rceil$ -byte string where $m$ is the key size in bits.
	<ul> <li>This is 32 bytes for Curve25519, computed as X25519(private_key, 9).</li> </ul>
	• This is 56 bytes for Curve448, computed as X448(private_key, 5).
Twisted Edwards Elliptic	PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_TWISTED_EDWARDS)
curve key pair	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.
Twisted Edwards Elliptic	PSA_KEY_TYPE_ECC_PUBLIC_KEY(PSA_ECC_FAMILY_TWISTED_EDWARDS)
curve public key	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.
Finite-field Diffie-Hellman key pair	PSA_KEY_TYPE_DH_KEY_PAIR(dh_family) where dh_family designates any Diffie-Hellman family.
	The key data is the representation of the private key $x$ as a big-endian byte string. The length of the byte string is the private key size in bytes, and leading zeroes are not stripped.
Finite-field Diffie-Hellman public key	PSA_KEY_TYPE_DH_PUBLIC_KEY(dh_family) where dh_family designates any Diffie-Hellman family.
	The key data 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.

Table 8 – continued from previous page

# 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 85.

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

- Hash algorithms on page 115
- MAC algorithms on page 136
- Cipher algorithms on page 152
- AEAD algorithms on page 179
- Key derivation algorithms on page 210
- Asymmetric signature algorithms on page 242
- Asymmetric encryption algorithms on page 264
- Key agreement algorithms on page 271

# 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.

```
0x0000001 - 0x7ffffff
```

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 - 0xfffffff

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 298 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 115 for a list of defined hash algorithms.

# PSA\_ALG\_IS\_MAC (macro)

Whether the specified algorithm is a MAC algorithm.

#define PSA\_ALG\_IS\_MAC(alg) /\* specification-defined value \*/

#### Parameters

alg

An algorithm identifier: 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 136 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 152 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 179 for a list of defined AEAD algorithms.

# PSA\_ALG\_IS\_SIGN (macro)

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

#define PSA\_ALG\_IS\_SIGN(alg) /\* specification-defined value \*/

#### Parameters

alg

An algorithm identifier: 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 algorithms on page 242 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 264 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 271 for a list of defined key agreement algorithms.

# PSA\_ALG\_IS\_KEY\_DERIVATION (macro)

Whether the specified algorithm is a key derivation algorithm.

#define PSA\_ALG\_IS\_KEY\_DERIVATION(alg) /\* specification-defined value \*/

#### Parameters

alg

An algorithm identifier: 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 210 for a list of defined key derivation algorithms.

# PSA\_ALG\_IS\_WILDCARD (macro)

Whether the specified algorithm encoding is a wildcard.

#define PSA\_ALG\_IS\_WILDCARD(alg) /\* specification-defined value \*/

#### Parameters

alg An algorithm identifier: 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\_ECDSA()
- PSA\_ALG\_HKDF()
- PSA\_ALG\_HKDF\_EXPAND()
- PSA\_ALG\_HKDF\_EXTRACT()
- PSA\_ALG\_HMAC()
- PSA\_ALG\_RSA\_OAEP()
- PSA\_ALG\_RSA\_PKCS1V15\_SIGN()
- PSA\_ALG\_RSA\_PSS()
- PSA\_ALG\_RSA\_PSS\_ANY\_SALT()
- PSA\_ALG\_TLS12\_PRF()
- PSA\_ALG\_TLS12\_PSK\_TO\_MS()
- PSA\_ALG\_PBKDF2\_HMAC()

# 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\_MM0\_ZIGBEE (macro)

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

#define PSA\_ALG\_AES\_MM0\_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)0x020000d)

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 SHAKE256 output.

#define PSA\_ALG\_SHAKE256\_512 ((psa\_algorithm\_t)0x02000015)

This is the prehashing for Ed448ph (see PSA\_ALG\_ED448PH).

SHAKE256 is defined in [FIPS202].

#### 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].

# 10.2.2 Single-part hashing functions

# psa\_hash\_compute (function)

Calculate the hash (digest) of a message.

#### 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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not a hash algorithm.</li> <li>input_length is too large for alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a hash algorithm.</li> <li>input_length is too large for the implementation.</li> </ul>
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.

#### 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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not a hash algorithm.</li> </ul>
	<ul> <li>input_length is too large for alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a hash algorithm.</li> </ul>
	<ul> <li>input_length is too large for the implementation.</li> </ul>
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.

Parameters
1 arameters

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 <pre>psa_algorithm_t such that PSA_ALG_IS_HASH(alg) is true.</pre>
Returns: psa_status_t	
PSA_SUCCESS	Success. The operation is now active.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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 24.

# psa\_hash\_update (function)

Add a message fragment to a multi-part hash operation.

#### 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:
	<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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.

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:
	<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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	

# 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.

#### 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:
	<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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
i alametero

	operation	Initialized hash operation.
Re	<b>eturns:</b> 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.

#### 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:
	<ul> <li>A sufficient output size is PSA_HASH_SUSPEND_OUTPUT_SIZE(alg) where alg is the algorithm that was used to set up the operation</li> </ul>
	<ul> <li>PSA_HASH_SUSPEND_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported hash algorithm.</li> </ul>
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:
	<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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 133 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.

<pre>psa_status_t psa_hash_resume(psa_hash_resume)</pre>	ash_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 <i>Hash suspend state format</i> on page 133.
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:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_INVALID_ARGUMENT	hash_state does not correspond to a valid hash suspend state. See <i>Hash suspend state format</i> on page 133 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.

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:
	<ul> <li>The source_operation state is not valid: it must be active.</li> <li>The target_operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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 134.

# 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 134.

# 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:

 $hash\_suspend\_state = algorithm \mid\mid input\_length \mid\mid hash\_state \mid\mid unprocessed\_input$ 

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.

# 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*.

# $unprocessed\_input$

# $0 \text{ to } (hash\_block\_size - 1) \text{ bytes}$

A partial block of unprocessed input data. This is between zero and  $hash_block_size - 1$  bytes of data, the length can be calculated by:

length(unprocessed\_input) = input\_length 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*.

# 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 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().

# 10.3.1 MAC algorithms

# PSA\_ALG\_HMAC (macro)

Macro to build an HMAC message-authentication-code algorithm from an underlying hash algorithm.

#define PSA\_ALG\_HMAC(hash\_alg) /\* specification-defined value \*/

#### Parameters

hash\_alg

A hash algorithm: a value of type psa\_algorithm\_t such that
PSA\_ALG\_IS\_HASH(hash\_alg) is true.

# Returns

The corresponding HMAC algorithm.

Unspecified if hash\_alg is not a supported hash algorithm.

# Description

For example, PSA\_ALG\_HMAC(PSA\_ALG\_SHA\_256) is HMAC-SHA-256.

The HMAC construction is defined in HMAC: Keyed-Hashing for Message Authentication [RFC2104].

# 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\_algA MAC algorithm: a value of type psa\_algorithm\_t such that<br/>PSA\_ALG\_IS\_MAC(mac\_alg) is true. This can be a truncated or<br/>untruncated MAC algorithm.mac\_lengthDesired length of the truncated MAC in bytes. This must be at most<br/>the untruncated length of the MAC and must be at least an<br/>implementation-specified minimum. The implementation-specified<br/>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.

```
#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 <pre>psa_algorithm_t such that PSA_ALG_IS_MAC(alg) is true. This can be a truncated or untruncated MAC algorithm.</pre>
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.

# 10.3.2 Single-part MAC functions

# psa\_mac\_compute (function)

Calculate the message authentication code (MAC) of a message.

(continues on next page)

(continued from previous page)

uint8_t * mac,
<pre>size_t mac_size,</pre>
<pre>size_t * mac_length);</pre>

#### 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.
	<ul> <li>PSA_MAC_MAX_SIZE evaluates to the maximum MAC size of any supported MAC algorithm.</li> </ul>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not a MAC algorithm.</li> <li>key is not compatible with alg.</li> <li>input_length is too large for alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a MAC algorithm.</li> </ul>
	<ul> <li>key is not supported for use with alg.</li> </ul>
	<ul> <li>input_length is too large for the implementation.</li> </ul>
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILUR	E

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.

#### 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 <pre>psa_algorithm_t such that PSA_ALG_IS_MAC(alg) is true.</pre>
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.
<b>Returns:</b> 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 <pre>psa_crypto_init().</pre>
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

# 10.3.3 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.

```
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 <pre>psa_algorithm_t such that PSA_ALG_IS_MAC(alg) is true.</pre>
<b>Returns:</b> psa_status_t	
PSA_SUCCESS	Success. The operation is now active.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>alg is not a MAC algorithm.</li> <li>key is not compatible with alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a MAC algorithm.</li> </ul>
	<ul> <li>key is not supported for use with alg.</li> </ul>
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 24.

## psa\_mac\_verify\_setup (function)

Set up a multi-part MAC verification operation.

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 <pre>psa_algorithm_t such that PSA_ALG_IS_MAC(alg) is true.</pre>
Returns: psa_status_t	
PSA_SUCCESS	Success. The operation is now active.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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.
DOL EDDOD THOUSET OTENT MEMORY	• key is not supported for use with arg.
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPI	
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  $\ensuremath{\texttt{psa\_mac\_verify\_setup}}$  () 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 24.

#### psa\_mac\_update (function)

Add a message fragment to a multi-part MAC operation.

#### 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.
Re	eturns: psa_status_t	
	PSA_SUCCESS	Success.
	PSA_ERROR_BAD_STATE	The following conditions can result in this error:
		<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
	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	

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\_MAC\_MAX\_SIZE evaluates to the maximum MAC size of any supported MAC algorithm.

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\_COMMUNICATION\_FAILURE
 PSA\_ERROR\_CORRUPTION\_DETECTED

 PSA\_ERROR\_STORAGE\_FAILURE
 ENCOMPLANCE

PSA\_ERROR\_DATA\_CORRUPT

mac\_length

PSA\_ERROR\_DATA\_INVALID

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.
                                     Buffer containing the expected MAC value.
  mac
                                     Size of the mac buffer in bytes.
  mac_length
Returns: psa_status_t
                                     Success. The expected MAC is identical to the actual MAC of the
  PSA_SUCCESS
                                     message.
                                     The following conditions can result in this error:
  PSA_ERROR_BAD_STATE
                                        • 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().
                                     The calculated MAC of the message does not match the value in mac.
  PSA_ERROR_INVALID_SIGNATURE
  PSA_ERROR_INSUFFICIENT_MEMORY
  PSA_ERROR_COMMUNICATION_FAILURE
  PSA_ERROR_CORRUPTION_DETECTED
  PSA_ERROR_STORAGE_FAILURE
  PSA_ERROR_DATA_CORRUPT
  PSA_ERROR_DATA_INVALID
```

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.
Re	eturns: 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 <pre>psa_crypto_init().</pre>
	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.

## **10.3.4 Support macros**

### PSA\_ALG\_IS\_HMAC (macro)

Whether the specified algorithm is an HMAC algorithm.

#define PSA\_ALG\_IS\_HMAC(alg) /\* specification-defined value \*/

### Parameters

alg

An algorithm identifier: 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 <pre>psa_algorithm_t</pre> such that
	PSA_ALG_IS_MAC(alg) is true.

## Returns

The MAC length for the specified algorithm with the specified key parameters.

 ${\ensuremath{\scriptscriptstyle 0}}$  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.

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().

# **10.4 Unauthenticated ciphers**

**Warning:** The unauthenticated cipher API is provided to implement legacy protocols and for use cases where the data integrity and authenticity is guaranteed by non-cryptographic means.

It is recommended that newer protocols use *Authenticated encryption with associated data* (AEAD) on page 178.

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().

# 10.4.1 Cipher algorithms

## PSA\_ALG\_STREAM\_CIPHER (macro)

The stream cipher mode of a stream cipher algorithm.

#define PSA\_ALG\_STREAM\_CIPHER ((psa\_algorithm\_t)0x04800100)

The underlying stream cipher is determined by the key type. The ARC4, 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  $1 \le n \le b$ , 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<sup>\*</sup> cipher mode without authentication.

#define PSA\_ALG\_CCM\_STAR\_NO\_TAG ((psa\_algorithm\_t)0x04c01300)

This is CCM<sup>\*</sup> as specified in *IEEE Standard for Low-Rate Wireless Networks* [IEEE-CCM] §7, with a tag length of 0. For CCM<sup>\*</sup> 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<sup>\*</sup> 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\_N0\_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

## 10.4.2 Single-part cipher functions

### psa\_cipher\_encrypt (function)

Encrypt a message using a symmetric cipher.

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:
	<ul> <li>A sufficient output size is         <pre>PSA_CIPHER_ENCRYPT_OUTPUT_SIZE(key_type, alg, input_length)         where key_type is the type of key.</pre> </li> <li>PSA_CIPHER_ENCRYPT_OUTPUT_MAX_SIZE(input_length) evaluates to         the maximum output size of any supported cipher encryption.</li> </ul>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not a cipher algorithm.</li> <li>key is not compatible with alg.</li> <li>The input_length is not valid for the algorithm and key type. For</li> </ul>
	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:
	<ul> <li>alg is not supported or is not a cipher algorithm.</li> </ul>
	<ul> <li>key is not supported for use with alg.</li> </ul>
	<ul> <li>input_length is too large for the implementation.</li> </ul>
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.

#### 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 <pre>psa_algorithm_t</pre> such that <pre>PSA_ALG_IS_CIPHER(alg)</pre> 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:
	<ul> <li>A sufficient output size is         <pre>PSA_CIPHER_DECRYPT_OUTPUT_SIZE(key_type, alg, input_length)         where key_type is the type of key.</pre> </li> <li>PSA_CIPHER_DECRYPT_OUTPUT_MAX_SIZE(input_length) evaluates to         the maximum output size of any supported cipher decryption.</li> </ul>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not a cipher algorithm.</li> <li>key is not compatible with alg.</li> <li>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.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a cipher algorithm.</li> <li>key is not supported for use with alg.</li> <li>input_length is too large for the implementation.</li> </ul>
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.

## 10.4.3 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.

Parameters	
operation	The operation object to set up. It must have been initialized as per the documentation for <pre>psa_cipher_operation_t</pre> 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.
<b>Returns:</b> psa_status_t	
PSA_SUCCESS	Success. The operation is now active.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>alg is not a cipher algorithm.</li> <li>key is not compatible with alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a cipher algorithm.</li> <li>key is not supported for use with alg.</li> </ul>
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA_ERROR_DATA_CORRUPT	
PSA_ERROR_DATA_INVALID	

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.
- 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 24.

### psa\_cipher\_decrypt\_setup (function)

Set the key for a multi-part symmetric decryption operation.

#### 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:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>alg is not a cipher algorithm.</li> <li>key is not compatible with alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a cipher algorithm.</li> <li>key is not supported for use with alg.</li> </ul>
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
PSA_ERROR_CORRUPTION_DETECTED	
PSA_ERROR_STORAGE_FAILURE	
PSA FRROR 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.
- 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 24.

## psa\_cipher\_generate\_iv (function)

Generate an initialization vector (IV) for a symmetric encryption operation.

<pre>psa_status_t psa_cipher_generate_iv(psa_cipher_operation_t * operation,</pre>			
	uint8_t * iv,		
	size_t iv_size,		
	<pre>size_t * iv_length);</pre>		
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:		
	<ul> <li>The cipher algorithm does not use an IV.</li> <li>The operation state is not valid: it must be active, with no IV set.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>		
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

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.

#### Parameters

	operation	Active cipher operation.
	iv	Buffer containing the IV to use.
	iv_length	Size of the IV in bytes.
Re	<b>eturns:</b> psa_status_t	
	PSA_SUCCESS	Success.
	PSA_ERROR_BAD_STATE	The following conditions can result in this error:
		<ul> <li>The cipher algorithm does not use an IV.</li> <li>The operation state is not valid: it must be an active cipher encrypt operation, with no IV set.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
	PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error:
		<ul> <li>The chosen algorithm does not use an IV.</li> <li>iv_length is not valid for the chosen algorithm.</li> </ul>
	PSA_ERROR_NOT_SUPPORTED	${\tt iv\_length}$ 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.

### 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.

R	<b>eturns:</b> 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:
		<ul> <li>The operation state is not valid: it must be active, with an IV set if required for the algorithm.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
	PSA_ERROR_BUFFER_TOO_SMALL	The size of the output buffer is too small.
		PSA_CIPHER_UPDATE_OUTPUT_SIZE() or
		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	

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.

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.
	<ul> <li>PSA_CIPHER_FINISH_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported cipher algorithm.</li> </ul>
output_length	On success, the number of bytes that make up the returned output.
<b>Returns:</b> 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.
	<ul> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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.
R	<b>eturns:</b> 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.

# 10.4.4 Support macros

PSA\_ALG\_IS\_STREAM\_CIPHER (macro)

Whether the specified algorithm is a stream cipher.

#define PSA\_ALG\_IS\_STREAM\_CIPHER(alg) /\* specification-defined value \*/

### Parameters

alg

An algorithm identifier: 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<sup>\*</sup> as both an AEAD and an unauthenticated cipher algorithm.

#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 <pre>psa_algorithm_t such that PSA_ALG_IS_CIPHER(alg) is true.</pre>
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.

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 <pre>psa_algorithm_t such that PSA_ALG_IS_CIPHER(alg) is true.</pre>
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.

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.

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 <pre>psa_algorithm_t</pre> 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().

# 10.5 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.

# 10.5.1 AEAD algorithms

# PSA\_ALG\_CCM (macro)

The Counter with CBC-MAC (CCM) authenticated encryption algorithm.

#define PSA\_ALG\_CCM ((psa\_algorithm\_t)0x05500100)

CCM is defined for block ciphers that have a 128-bit block size. The underlying block cipher is determined by the key type.

To use PSA\_ALG\_CCM with a multi-part AEAD operation, the application must call psa\_aead\_set\_lengths() before providing the nonce, the additional data and plaintext to the operation.

CCM requires a nonce of between 7 and 13 bytes in length. The length of the nonce 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  $L = 15 \text{nonce\_length}$ . 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  $L \ge 2$ , 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<sup>\*</sup>, 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\_P0LY1305 (macro)

The XChaCha20-Poly1305 AEAD algorithm.

#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\_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 <pre>psa_algorithm_t such that PSA_ALG_IS_AEAD(aead_alg) is true.</pre>
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.

```
#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 <pre>psa_algorithm_t such that PSA_ALG_IS_AEAD(aead_alg) is true.</pre>
<pre>min_tag_length</pre>	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.

# 10.5.2 Single-part AEAD functions

### psa\_aead\_encrypt (function)

Process an authenticated encryption operation.

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 <pre>psa_algorithm_t</pre> such that <pre>PSA_ALG_IS_AEAD(alg)</pre> 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:
	<ul> <li>A sufficient output size is         PSA_AEAD_ENCRYPT_OUTPUT_SIZE(key_type, alg, plaintext_length)         where key_type is the type of key.     </li> </ul>

	<ul> <li>PSA_AEAD_ENCRYPT_OUTPUT_MAX_SIZE(plaintext_length) evaluates to the maximum ciphertext size of any supported AEAD encryption.</li> </ul>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not an AEAD algorithm.</li> <li>key is not compatible with alg.</li> <li>nonce_length is not valid for use with alg and key.</li> <li>additional_data_length or plaintext_length are too large for alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an AEAD algorithm.</li> <li>key is not supported for use with alg.</li> <li>nonce_length is not supported for use with alg and key.</li> <li>additional_data_length or plaintext_length are too large for the implementation.</li> </ul>
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILURE	
DOL EDDOD CODDUDITON DETECTED	

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.

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 <pre>psa_algorithm_t</pre> such that <pre>PSA_ALG_IS_AEAD(alg)</pre> 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:
	<ul> <li>A sufficient output size is         <pre>PSA_AEAD_DECRYPT_OUTPUT_SIZE(key_type, alg,             ciphertext_length) where key_type is the type of key.</pre> </li> <li>PSA_AEAD_DECRYPT_OUTPUT_MAX_SIZE(ciphertext_length) evaluates         to the maximum plaintext size of any supported AEAD         decryption.</li> </ul>
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. The library requires initializing by a call to psa\_crypto\_init(). PSA\_ERROR\_BAD\_STATE key is not a valid key identifier. PSA\_ERROR\_INVALID\_HANDLE The key does not have the PSA\_KEY\_USAGE\_DECRYPT flag, or it does not PSA\_ERROR\_NOT\_PERMITTED permit the requested algorithm. The ciphertext is not authentic. PSA\_ERROR\_INVALID\_SIGNATURE The size of the plaintext buffer is too small. PSA\_ERROR\_BUFFER\_TOO\_SMALL PSA\_AEAD\_DECRYPT\_OUTPUT\_SIZE() or PSA\_AEAD\_DECRYPT\_OUTPUT\_MAX\_SIZE() can be used to determine a sufficient buffer size. The following conditions can result in this error: PSA\_ERROR\_INVALID\_ARGUMENT • 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. The following conditions can result in this error: PSA\_ERROR\_NOT\_SUPPORTED • 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

# 10.5.3 Multi-part AEAD operations

PSA\_ERROR\_DATA\_INVALID

**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.

```
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.
<b>Returns:</b> psa_status_t	
PSA_SUCCESS	Success. The operation is now active.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>alg is not an AEAD algorithm.</li> <li>key is not compatible with alg.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an AEAD algorithm.</li> </ul>
	<ul> <li>key is not supported for use with alg.</li> </ul>

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 24.

### 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
                                     The operation object to set up. It must have been initialized as per
   operation
                                     the documentation for psa_aead_operation_t and not yet in use.
                                     Identifier of the key to use for the operation. It must remain valid
   key
                                     until the operation terminates. It must permit the usage
                                     PSA_KEY_USAGE_DECRYPT.
                                     The AEAD algorithm to compute: a value of type psa_algorithm_t
   alg
                                     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.
                                     The following conditions can result in this error:
   PSA_ERROR_INVALID_ARGUMENT
                                        • alg is not an AEAD algorithm.
                                        • key is not compatible with alg.
                                     The following conditions can result in this error:
   PSA_ERROR_NOT_SUPPORTED
                                        • 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 24.

### psa\_aead\_set\_lengths (function)

Declare the lengths of the message and additional data for AEAD.

<pre>psa_status_t psa_aead_set_lengths(psa_aead_operation_t * operation,</pre>	
<pre>size_t ad_length,</pre>	
<pre>size_t plaintext_length);</pre>	

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.

R	Returns: psa_status_t		
	PSA_SUCCESS	Success.	
	PSA_ERROR_BAD_STATE	The following conditions can result in this error:	
		<ul> <li>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.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>	
	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	<pre>psa_aead_generate_nonce(psa_aead_operation_t * operation,</pre>
	uint8_t * nonce,
	<pre>size_t nonce_size,</pre>
	size t * nonce length):

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:
	<ul> <li>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.</li> <li>PSA_AEAD_NONCE_MAX_SIZE evaluates to a sufficient output size for any supported AEAD algorithm.</li> </ul>

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:			
	<ul> <li>The operation state is not valid: it must be an active AEAD encryption operation, with no nonce set.</li> <li>The operation state is not valid: this is an algorithm which requires psa_aead_set_lengths() to be called before setting the nonce.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>			
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.

**Parameters** Active AEAD operation. operation Buffer containing the nonce to use. nonce Size of the nonce in bytes. This must be a valid nonce size for the nonce\_length 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(). nonce\_length is not valid for the chosen algorithm. PSA\_ERROR\_INVALID\_ARGUMENT nonce\_length is not supported for use with the operation's algorithm PSA\_ERROR\_NOT\_SUPPORTED 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 cor siz	<pre>a_aead_operation_t * operation, nst uint8_t * input, ze_t input_length);</pre>
Parameters	
operation	Active AEAD operation.
input	Buffer containing the fragment of additional data.
input_length	Size of the input buffer in bytes.
<b>Returns:</b> 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:
	<ul> <li>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.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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.

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.
	<ul> <li>PSA_AEAD_UPDATE_OUTPUT_MAX_SIZE(input_length) evaluates to the maximum output size of any supported AEAD algorithm.</li> </ul>
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:
	<ul> <li>The operation state is not valid: it must be active, have a nonce set, and have lengths set if required by the algorithm.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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	<ul> <li>The following conditions can result in this error:</li> <li>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().</li> <li>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.</li> </ul>
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  $\ensuremath{\texttt{psa\_aead\_generate\_nonce}}\)$  or  $\ensuremath{\texttt{psa\_aead\_set\_nonce}}\).$
- 3. Call  $\ensuremath{\texttt{psa}\_aead\_update\_ad}(\ensuremath{)}$  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.

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.
	<ul> <li>PSA_AEAD_FINISH_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported AEAD algorithm.</li> </ul>
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 ( $\star$ tag_length) bytes of tag contain the authentication tag.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be an active encryption operation with a nonce set.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>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().</li> <li>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().</li> </ul>
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.

<pre>psa_status_t psa_aead_verify(psa_a</pre>	ead_operation_t * operation,
uint8	_t * plaintext,
size_	t plaintext_size,
size_	t * plaintext_length,
size	t  tag
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.
	<ul> <li>PSA_AEAD_VERIFY_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported AEAD algorithm.</li> </ul>
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:
	<ul> <li>The operation state is not valid: it must be an active decryption operation with a nonce set.</li> </ul>
	<ul> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>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().</li> <li>Incomplete ciphertext: the total length of input to psa_aead_update() is less than the plaintext length that was</li> </ul>

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.

# 10.5.4 Support macros

### PSA\_ALG\_IS\_AEAD\_ON\_BLOCK\_CIPHER (macro)

Whether the specified algorithm is an AEAD mode on a block cipher.

#define PSA\_ALG\_IS\_AEAD\_ON\_BLOCK\_CIPHER(alg) /\* specification-defined value \*/

#### Parameters

alg An algorithm identifier: 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 */
```

key_type	A symmetric key type that is compatible with algorithm alg.
alg	An AEAD algorithm: a value of type <pre>psa_algorithm_t such that PSA_ALG_IS_AEAD(alg) is true.</pre>
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 */
```

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 <pre>psa_algorithm_t</pre> 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

### 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 */
```

key_type	A symmetric key type that is compatible with algorithm alg.
alg	An AEAD algorithm: a value of type <pre>psa_algorithm_t</pre> 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().

# 10.6 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() 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.

 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.

# 10.6.1 Key derivation algorithms

### PSA\_ALG\_HKDF (macro)

Macro to build an HKDF algorithm.

#define PSA\_ALG\_HKDF(hash\_alg) /\* specification-defined value \*/

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.

#define PSA\_ALG\_HKDF\_EXTRACT(hash\_alg) /\* specification-defined value \*/

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.

#define PSA\_ALG\_HKDF\_EXPAND(hash\_alg) /\* specification-defined value \*/

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.

```
#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:

 $K_i = \mathsf{HMAC}(K_{IN}, [i]_4 \mid\mid Label \mid\mid \texttt{0x00} \mid\mid Context \mid\mid [L]_4), \quad \text{for } i = 1, 2, 3, \dots$ 

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.

#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*<sub>IN</sub>. 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  $K_1 \parallel K_2 \parallel K_3 \parallel ...$  is computed as:

$$\begin{split} K_0 &= \mathsf{CMAC}(K_{IN}, Label \mid\mid \texttt{0x00} \mid\mid Context \mid\mid [L]_4 \ ) \\ K_i &= \mathsf{CMAC}(K_{IN}, [i]_4 \mid\mid Label \mid\mid \texttt{0x00} \mid\mid Context \mid\mid [L]_4 \mid\mid K_0), \quad \text{for } i = 1, 2, 3, \ldots \end{split}$$

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 \*/
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\_T0\_MS (macro)

Macro to build a TLS-1.2 PSK-to-MasterSecret algorithm.

#define PSA\_ALG\_TLS12\_PSK\_T0\_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\_T0\_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\_T0\_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().
- $\bullet$  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\_T0\_PMS (macro)

The TLS 1.2 ECJPAKE-to-PMS key-derivation algorithm.

#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 the extension to the Crypto API defined in *PSA Certified Crypto API 1.2 PAKE Extension* [PSA-PAKE].

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. These are defined in the PAKE extension API, see [PSA-PAKE].

The 32-byte output has to be read in a single call to either psa\_key\_derivation\_output\_bytes() or psa\_key\_derivation\_output\_key(). 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\_PBKDF2\_HMAC (macro)

Macro to build a PBKDF2-HMAC password-hashing or key-stretching algorithm.

#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 pseudo-random 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.

#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 pseudo-random 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)

# 10.6.2 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().

## PSA\_KEY\_DERIVATION\_INPUT\_OTHER\_SECRET (macro)

A high-entropy additional secret input for key derivation.

```
#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.

#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().

## 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.

#define PSA\_KEY\_DERIVATION\_INPUT\_COST /\* implementation-defined value \*/

This must be a direct input, passed to psa\_key\_derivation\_input\_integer().

# 10.6.3 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.

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 <pre>psa_algorithm_t such that PSA_ALG_IS_KEY_DERIVATION(alg) is true.</pre>
	• A key agreement and 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:
	<ul> <li>The operation state is not valid: it must be inactive.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_INVALID_ARGUMENT	alg is neither a key derivation algorithm, nor a key agreement and derivation algorithm.
PSA_ERROR_NOT_SUPPORTED	alg is not supported or is not a key derivation algorithm, or a key agreement and 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 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(). 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 24.

## psa\_key\_derivation\_get\_capacity (function)

Retrieve the current capacity of a key derivation operation.

Parameters	
operation	The operation to query.
capacity	On success, the capacity of the operation.
<b>Returns:</b> 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:
	<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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() 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.

#### 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.
<b>Returns:</b> psa_status_t	
PSA_SUCCESS	Success.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be active.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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.

	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.
Re	eturns: psa_status_t	
	PSA_SUCCESS	Success.
	PSA_ERROR_BAD_STATE	The following conditions can result in this error:
		<ul> <li>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.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
	PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error:
		<ul> <li>step is not compatible with the operation's algorithm.</li> </ul>
		<ul> <li>step does not permit direct inputs.</li> </ul>
		<ul> <li>data_length is too small or too large for step in this particular algorithm.</li> </ul>
	PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
		<ul> <li>step is not supported with the operation's algorithm.</li> </ul>
		<ul> <li>data_length is is not supported for step in this particular algorithm.</li> </ul>
	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.

<pre>psa_status_t psa_key_derivation_ing</pre>	<pre>put_integer(psa_key_derivation_operation_t * operation,</pre>
	<pre>psa_key_derivation_step_t step,</pre>
	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:
	<ul> <li>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.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error:
	<ul> <li>step is not compatible with the operation's algorithm.</li> <li>step does not permit numerical inputs.</li> <li>value is not valid for step in the operation's algorithm.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>step is not supported with the operation's algorithm.</li> <li>value is not supported for step in the operation's algorithm.</li> </ul>
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.

<pre>psa_status_t psa_key_derivation_input_key(psa_key_derivation_operation_t * operation,</pre>	
	<pre>psa_key_derivation_step_t step,</pre>
	<pre>psa_key_id_t key);</pre>
Parameters	
operation	The key derivation operation object to use. It must have been set up with <pre>psa_key_derivation_setup()</pre> 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.
	<ul> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_INVALID_HANDLE	key is not a valid key identifier.
PSA_ERROR_NOT_PERMITTED	The key has neither the PSA_KEY_USAGE_DERIVE nor the PSA_KEY_USAGE_VERIFY_DERIVATION usage flag, or it does not permit the operation's algorithm.
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error:
	<ul> <li>step is not compatible with the operation's algorithm.</li> </ul>

• 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() 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() if each input was either a direct input, a key with usage flag PSA\_KEY\_USAGE\_DERIVE, or a key with usage flag PSA\_KEY\_USAGE\_VERIFY\_DERIVATION.
- psa\_key\_derivation\_verify\_key() under the same conditions as psa\_key\_derivation\_verify\_bytes().

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
                                     The key derivation operation object to read from.
   operation
                                     Buffer where the output will be written.
   output
                                     Number of bytes to output.
  output_length
Returns: psa_status_t
  PSA SUCCESS
                                     Success. The first output_length bytes of output contain the derived
                                     data.
                                     The following conditions can result in this error:
   PSA_ERROR_BAD_STATE
                                        • 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().
                                     One of the inputs was a key whose policy did not permit
   PSA_ERROR_NOT_PERMITTED
                                     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.

<pre>psa_status_t psa_key_derivation_</pre>	<pre>output_key(const psa_key_attributes_t * attributes,</pre>
Parameters	
attributes	The attributes for the new key. This function uses the attributes as follows:
	<ul> <li>The key type is required. It cannot be an asymmetric public key.</li> <li>The key size is required. It must be a valid size for the key type.</li> <li>The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see <i>Permitted algorithms</i> on page 85. If the key type to be created is PSA_KEY_TYPE_PASSWORD_HASH, then the permitted-algorithm policy must be the same as the current operation's algorithm.</li> <li>The key usage flags define what operations are permitted with the key, see <i>Key usage flags</i> on page 87.</li> <li>The key lifetime and identifier are required for a persistent key.</li> </ul>
	Note: This is an input parameter: it is not updated with the final key attributes. The final attributes of the new key can be queried by calling psa_get_key_attributes() with the key's identifier.
operation	The key derivation operation object to read from.
key	On success, an identifier for the newly created key. PSA_KEY_ID_NULL on failure.
Returns: psa_status_t	
PSA_SUCCESS	Success. If the key is persistent, the key material and the key's metadata have been saved to persistent storage.
PSA_ERROR_BAD_STATE	The following conditions can result in this error:
	<ul> <li>The operation state is not valid: it must be active, with all required input steps complete.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_NOT_PERMITTED	The following conditions can result in this error:
	• The PSA_KEY_DERIVATION_INPUT_SECRET input step was neither provided through a key, nor the result of a key agreement.

	<ul> <li>One of the inputs was a key whose policy did not permit PSA_KEY_USAGE_DERIVE.</li> </ul>
	<ul> <li>The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.</li> </ul>
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:
	<ul> <li>No key is generated.</li> </ul>
	<ul> <li>The operation's capacity is set to zero.</li> </ul>
PSA_ERROR_INVALID_ARGUMENT	The following conditions can result in this error:
	<ul> <li>The key type is invalid, or is an asymmetric public key type.</li> <li>The key type is PSA_KEY_TYPE_PASSWORD_HASH, and the permitted-algorithm policy is not the same as the current operation's algorithm.</li> <li>The key size is not valid for the key type. Implementations must reject an attempt to derive a key of size 0.</li> <li>The key lifetime is invalid.</li> <li>The key identifier is not valid for the key lifetime.</li> <li>The key usage flags include invalid values.</li> <li>The key's permitted-usage algorithm is invalid.</li> <li>The key attributes, as a whole, are invalid.</li> </ul>
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. Table 9 describes the required key derivation procedures for standard key derivation algorithms. Implementations can use other methods for implementation-specific algorithms.

Key type	Key type details and derivation procedure
AES	PSA_KEY_TYPE_AES
ARC4	PSA_KEY_TYPE_ARC4
ARIA	PSA_KEY_TYPE_ARIA
CAMELLIA	PSA_KEY_TYPE_CAMELLIA
ChaCha20	PSA_KEY_TYPE_CHACHA20
SM4	PSA_KEY_TYPE_SM4
Secrets for derivation	PSA_KEY_TYPE_DERIVE
HMAC	PSA_KEY_TYPE_HMAC
Password hashes	PSA_KEY_TYPE_PASSWORD_HASH
	For key types for which the key is an arbitrary sequence of bytes of a given size, this function is functionally equivalent to calling psa_key_derivation_output_bytes() and passing the resulting output to psa_import_key(). However, this function has a security benefit: if the implementation provides an isolation boundary then the key material is not exposed outside the isolation boundary. As a consequence, for these key types, this function always consumes exactly key_bits/8 bytes from the operation.
DES	PSA_KEY_TYPE_DES, 64 bits.
	This function generates a key using the following process:
	1. Draw an 8-byte string.
	2. Set/clear the parity bits in each byte.
	<ol> <li>If the result is a forbidden weak key, discard the result and return to step 1.</li> <li>Output the string.</li> </ol>
2-key 3DES	PSA_KEY_TYPE_DES, 192 bits.
3-key 3DES	PSA_KEY_TYPE_DES, 128 bits.
	The two or three keys are generated by repeated application of the process used to generate a DES key.
	For example, for 3-key 3DES, if the first 8 bytes specify a weak key and the next 8 bytes do not, discard the first 8 bytes, use the next 8 bytes as the first key, and continue reading output from the operation to derive the other two keys.

 Table 9 Standard key derivation process

#### continues on next page

Key type	Key type details and derivation procedure
Finite-field Diffie-Hellman keys	PSA_KEY_TYPE_DH_KEY_PAIR(dh_family) where dh_family designates any Diffie-Hellman family.
ECC keys on a Weierstrass elliptic curve	PSA_KEY_TYPE_ECC_KEY_PAIR(ecc_family) where ecc_family designates a Weierstrass curve family.
	These key types require the generation of a private key $d \in [1, N - 1]$ , where N is the boundary of the private key domain: N is the prime p for Diffie-Hellman, or the order of the curve's base point for ECC.
	Let $m$ be the bit size of $N$ , such that $2^{m-1} \le N < 2^m$ . 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.
	<ol> <li>Convert the string to integer k by decoding it as a big-endian byte-string.</li> </ol>
	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 <i>Key-Pair Generation by Testing Candidates</i> in the following publications:
	<ul> <li>NIST Special Publication 800-56A: Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography [SP800-56A] §5.6.1.1.4 for Diffie-Hellman keys.</li> </ul>
	• [SP800-56A] §5.6.1.2.2 or FIPS Publication 186-4: Digital Signature Standard (DSS) [FIPS186-4] §B.4.2 for elliptic curve keys.
ECC keys on a Montgomery	PSA_KEY_TYPE_ECC_KEY_PAIR(PSA_ECC_FAMILY_MONTGOMERY)
elliptic curve	This function always draws a byte string whose length is determined by the curve, and sets the mandatory bits accordingly. That is:
	• Curve25519 (PSA_ECC_FAMILY_MONTGOMERY, 255 bits): draw a 32-byte string and process it as specified in <i>Elliptic Curves for Security</i> [RFC7748] §5.
	• Curve448 (PSA_ECC_FAMILY_MONTGOMERY, 448 bits): draw a 56-byte string and process it as specified in [RFC7748] §5.
Other key types	This includes PSA_KEY_TYPE_RSA_KEY_PAIR.
	The way in which the operation output is consumed is implementation-defined.

 Table 9 - continued from previous page

For algorithms that take an input step PSA\_KEY\_DERIVATION\_INPUT\_SECRET, the input to that step must be

provided with psa\_key\_derivation\_input\_key(). Future versions of this specification might include additional restrictions on the derived key based on the attributes and strength of the secret key.

## psa\_key\_derivation\_verify\_bytes (function)

Compare output data from a key derivation operation to an expected value.

<pre>psa_status_t psa_key_derivation_verify_bytes(psa_key_derivation_operation_t * operation,</pre>	
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:
	<ul> <li>The operation state is not valid: it must be active, with all required input steps complete.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_NOT_PERMITTED	One of the inputs is a key whose policy permits neither PSA_KEY_USAGE_DERIVE nor PSA_KEY_USAGE_VERIFY_DERIVATION.
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 the key's policy does not permit output of the bytes.

## 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.

<pre>psa_status_t psa_key_derivation_verify_key(psa_key_derivation_operation_t * operation,</pre>		
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().	

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:
	<ul> <li>The operation state is not valid: it must be active, with all required input steps complete.</li> </ul>
	<ul> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
PSA_ERROR_INVALID_HANDLE	expected is not a valid key identifier.
PSA_ERROR_NOT_PERMITTED	The following conditions can result in this error:
	• The expected key does not have the PSA_KEY_USAGE_VERIFY_DERIVATION flag, or it does not permit the requested algorithm.
	<ul> <li>One of the inputs is a key whose policy permits neither PSA_KEY_USAGE_DERIVE nor PSA_KEY_USAGE_VERIFY_DERIVATION.</li> </ul>
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

#### 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.
<b>Returns:</b> 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.

## **10.6.4 Support macros**

## PSA\_ALG\_IS\_KEY\_DERIVATION\_STRETCHING (macro)

Whether the specified algorithm is a key-stretching or password-hashing algorithm.

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

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)).

#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)).

#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).

```
#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\_T0\_MS (macro)

Whether the specified algorithm is a TLS-1.2 PSK to MS algorithm.

#define PSA\_ALG\_IS\_TLS12\_PSK\_T0\_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.

#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\_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\_T0\_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\_T0\_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\_T0\_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\_T0\_MS\_PSK\_MAX\_SIZE with a value greater than or equal to 64.

## PSA\_TLS12\_ECJPAKE\_T0\_PMS\_0UTPUT\_SIZE (macro)

The size of the output from the TLS 1.2 ECJPAKE-to-PMS key-derivation algorithm, in bytes.

#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.

# **10.7** Asymmetric signature

There are two pairs of single-part functions for asymmetric signature:

- The signature and verification functions psa\_sign\_message() and psa\_verify\_message() take a message as one of their inputs and perform a hash-and-sign algorithm.
- The functions psa\_sign\_hash() and psa\_verify\_hash() take a message hash as one of their inputs. This is useful for signing pre-computed hashes, or for implementing hash-and-sign using a multi-part hash operation before signing the resulting hash. To determine which hash algorithm to use, call the macro PSA\_ALG\_GET\_HASH() on the corresponding signature algorithm.

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

## 10.7.1 Asymmetric signature algorithms

## PSA\_ALG\_RSA\_PKCS1V15\_SIGN (macro)

The RSA PKCS#1 v1.5 message signature scheme, with hashing.

#define PSA\_ALG\_RSA\_PKCS1V15\_SIGN(hash\_alg) /\* specification-defined value \*/

#### Parameters

hash\_alg

A hash algorithm: 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 algorithm can be used with both the message and hash signature functions.

This signature scheme is defined by PKCS #1: RSA Cryptography Specifications Version 2.2 [RFC8017] §8.2 under the name RSASSA-PKCS1-v1\_5.

When used with  $psa_sign_hash()$  or  $psa_verify_hash()$ , the provided hash parameter is used as H from step 2 onwards in the message encoding algorithm EMSA-PKCS1-V1\_5-ENCODE() in [RFC8017] §9.2. H is 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 algorithm can be only used with the psa\_sign\_hash() and psa\_verify\_hash() functions.

This signature scheme is defined by PKCS #1: RSA Cryptography Specifications Version 2.2 [RFC8017] §8.2 under the name RSASSA-PKCS1-v1\_5.

The hash parameter to  $psa\_sign\_hash()$  or  $psa\_verify\_hash()$  is used as T from step 3 onwards in the message encoding algorithm EMSA-PKCS1-V1\_5-ENCODE() in [RFC8017] §9.2. T is 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 \*/

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 algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the signature scheme defined by [RFC8017] §8.1 under the name RSASSA-PSS, with the following options:

- The mask generation function is MGF1 defined by [RFC8017] Appendix B.
- 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.

#define PSA\_ALG\_RSA\_PSS\_ANY\_SALT(hash\_alg) /\* specification-defined value \*/

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 algorithm can be used with both the message and hash signature functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the signature scheme defined by [RFC8017] §8.1 under the name RSASSA-PSS, with the following options:

- The mask generation function is MGF1 defined by [RFC8017] Appendix B.
- 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\_ECDSA (macro)

The randomized ECDSA signature scheme, with hashing.

#define PSA\_ALG\_ECDSA(hash\_alg) /\* specification-defined value \*/

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 algorithm can be used with both the message and hash signature functions.

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.

#### Note:

When based on the same hash algorithm, the verification operations for PSA\_ALG\_ECDSA and PSA\_ALG\_DETERMINISTIC\_ECDSA are identical. A signature created using PSA\_ALG\_ECDSA can be verified with the same key using either PSA\_ALG\_ECDSA or PSA\_ALG\_DETERMINISTIC\_ECDSA. Similarly, a signature created using PSA\_ALG\_DETERMINISTIC\_ECDSA can be verified with the same key using either PSA\_ALG\_DETERMINISTIC\_ECDSA can be verified with the same key using either PSA\_ALG\_DETERMINISTIC\_ECDSA.

In particular, it is impossible to determine whether a signature was produced with deterministic ECDSA or with randomized ECDSA: it is only possible to verify that a signature was made with ECDSA with the private key corresponding to the public key used for the verification.

This signature scheme is defined by SEC 1: Elliptic Curve Cryptography [SEC1], and also by Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA) [X9-62], with a random per-message secret number k.

The representation of the signature as a byte string consists of the concatenation of the signature values r and s. Each of r and s is encoded as an N-octet string, where N is the length of the base point of the curve in octets. Each value is represented in big-endian order, with the most significant octet first.

## 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 algorithm can be only used with the psa\_sign\_hash() and psa\_verify\_hash() functions.

This algorithm is randomized: each invocation returns a different, equally valid signature.

This is the same signature scheme as PSA\_ALG\_ECDSA(), but without specifying a hash algorithm, and skipping the message hashing operation.

This algorithm is only recommended to sign or verify a sequence of bytes that are an already-calculated hash. Note that the input is padded with zeros on the left or truncated on the right as required to fit the curve size.

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 algorithm can be used with both the message and hash signature functions.

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().

## Note:

When based on the same hash algorithm, the verification operations for PSA\_ALG\_ECDSA and PSA\_ALG\_DETERMINISTIC\_ECDSA are identical. A signature created using PSA\_ALG\_ECDSA can be verified with the same key using either PSA\_ALG\_ECDSA or PSA\_ALG\_DETERMINISTIC\_ECDSA. Similarly, a signature created using PSA\_ALG\_DETERMINISTIC\_ECDSA can be verified with the same key using either PSA\_ALG\_DETERMINISTIC\_ECDSA can be verified with the same key using either PSA\_ALG\_DETERMINISTIC\_ECDSA.

In particular, it is impossible to determine whether a signature was produced with deterministic ECDSA or with randomized ECDSA: it is only possible to verify that a signature was made with ECDSA with the private key corresponding to the public key used for the verification.

## 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\_PURE\_EDDSA (macro)

Edwards-curve digital signature algorithm without prehashing (PureEdDSA), using standard parameters.

#define PSA\_ALG\_PURE\_EDDSA ((psa\_algorithm\_t) 0x06000800)

This algorithm can be only used with the psa\_sign\_message() and psa\_verify\_message() functions.

This is the PureEdDSA digital signature algorithm defined by *Edwards-Curve Digital Signature Algorithm* (*EdDSA*) [RFC8032], using standard parameters.

PureEdDSA requires an elliptic curve key on a twisted Edwards curve. 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 an empty string as the context. The output signature is a 114-byte string: the concatenation of *R* and *S* as defined by [RFC8032] §5.2.6.

#### Note:

Contexts are not supported in the current version of this specification because there is no suitable signature interface that can take the context as a parameter. A future version of this specification may add suitable functions and extend this algorithm to support contexts.

#### 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 with psa\_sign\_hash() and psa\_verify\_hash().

The signature produced by HashEdDSA is distinct from that produced by PureEdDSA.

#### 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 prehashing (HashEdDSA), using the Edwards25519 curve.

#define PSA\_ALG\_ED25519PH ((psa\_algorithm\_t) 0x0600090B)

This algorithm can be used with both the message and hash signature functions.

This computes the Ed25519ph algorithm as specified in *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032] §5.1, and requires an Edwards25519 curve key. An empty string is used as the context. The prehash function is SHA-512.

When used with psa\_sign\_hash() or psa\_verify\_hash(), the provided hash parameter is the SHA-512 message digest.

#### Implementation note

When used with  $psa_sign_hash()$  or  $psa_verify_hash()$ , the hash parameter to the call should be used as 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() 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().

Verifying a signature is similar, using psa\_verify\_message() or psa\_verify\_hash() instead of the signature function.

## 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 prehashing (HashEdDSA), using the Edwards448 curve.

#define PSA\_ALG\_ED448PH ((psa\_algorithm\_t) 0x06000915)

This algorithm can be used with both the message and hash signature functions.

This computes the Ed448ph algorithm as specified in *Edwards-Curve Digital Signature Algorithm (EdDSA)* [RFC8032] §5.2, and requires an Edwards448 curve key. An empty string is used as the context. The prehash function is the first 64 bytes of the output from SHAKE256.

When used with psa\_sign\_hash() or psa\_verify\_hash(), the provided hash parameter is the truncated SHAKE256 message digest.

## Implementation note

When used with psa\_sign\_hash() or psa\_verify\_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() 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().

Verifying a signature is similar, using psa\_verify\_message() or psa\_verify\_hash() instead of the signature function.

### 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)
```

# 10.7.2 Asymmetric signature functions

## psa\_sign\_message (function)

Sign a message with a private key. For hash-and-sign algorithms, this includes the hashing step.

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 <pre>psa_algorithm_t</pre> such that <pre>PSA_ALG_IS_SIGN_MESSAGE(alg)</pre> 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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not an asymmetric signature algorithm.</li> <li>key is not an asymmetric key pair, that is compatible with alg.</li> <li>input_length is too large for the algorithm and key type.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an asymmetric signature algorithm.</li> <li>key is not supported for use with alg.</li> <li>input_length is too large for the implementation.</li> </ul>
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

# Note:

To perform a multi-part hash-and-sign signature algorithm, first use a multi-part hash operation and then pass the resulting hash to psa\_sign\_hash(). PSA\_ALG\_GET\_HASH(alg) can be used to determine the hash algorithm to use.

# psa\_verify\_message (function)

Verify the signature of a message with a public key. For hash-and-sign algorithms, this includes the hashing step.

<pre>psa_status_t psa_verify_message</pre>	(psa_key_id_t key,
	<pre>psa_algorithm_t alg,</pre>
	<pre>const uint8_t * input,</pre>
	<pre>size_t input_length, const uint&amp; t * signature</pre>
	<pre>size_t signature_length);</pre>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not an asymmetric signature algorithm.</li> <li>key is not a public key or an asymmetric key pair, that is compatible with alg.</li> <li>input_length is too large for the algorithm and key type.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an asymmetric signature algorithm.</li> <li>key is not supported for use with alg.</li> <li>input_length is too large for the implementation.</li> </ul>
PSA_ERROR_INSUFFICIENT_MEMORY	
PSA_ERROR_COMMUNICATION_FAILL	IRE

PSA\_ERROR\_CORRUPTION\_DETECTED PSA\_ERROR\_STORAGE\_FAILURE PSA\_ERROR\_DATA\_CORRUPT

PSA\_ERROR\_DATA\_INVALID

#### Description

# Note:

To perform a multi-part hash-and-sign signature verification algorithm, first use a multi-part hash operation to hash the message and then pass the resulting hash to psa\_verify\_hash(). PSA\_ALG\_GET\_HASH(alg) can be used to determine the hash algorithm to use.

# psa\_sign\_hash (function)

Sign an already-calculated hash with a private key.

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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not an asymmetric signature algorithm.</li> <li>key is not an asymmetric key pair, that is compatible with alg.</li> <li>hash_length is not valid for the algorithm and key type.</li> <li>hash is not a valid input value for the algorithm and key type.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an asymmetric signature algorithm.</li> <li>key is not supported for use with alg.</li> </ul>
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

With most signature algorithms that follow the hash-and-sign paradigm, the hash input to this function is the hash of the message to sign. The algorithm used to compute this hash is encoded in the signature algorithm. For such algorithms, hash\_length must equal the length of the hash output, and the following condition is true:

hash\_length == PSA\_HASH\_LENGTH(PSA\_ALG\_GET\_HASH(alg))

The current version of this specification defines the following signature algorithms with this property: PSA\_ALG\_RSA\_PKCS1V15\_SIGN, PSA\_ALG\_RSA\_PSS, PSA\_ALG\_ECDSA, PSA\_ALG\_DETERMINISTIC\_ECDSA, PSA\_ALG\_ED25519PH, and PSA\_ALG\_ED448PH.

Some hash-and-sign mechanisms apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. The current version of this specification defines one such signature

algorithm: PSA\_ALG\_RSA\_PKCS1V15\_SIGN\_RAW.

#### Note:

To perform a hash-and-sign signature algorithm, the hash must be calculated before passing it to this function. This can be done by calling psa\_hash\_compute() or with a multi-part hash operation. The correct hash algorithm to use can be determined using PSA\_ALG\_GET\_HASH().

Alternatively, to hash and sign a message in a single call, use psa\_sign\_message().

#### psa\_verify\_hash (function)

Verify the signature of a hash or short message using a public key.

<pre>&gt;sa_status_t psa_verify_hash(psa_key_id_t key,</pre>
<pre>psa_algorithm_t alg,</pre>
<pre>const uint8_t * hash,</pre>
<pre>size_t hash_length,</pre>
<pre>const uint8_t * signature,</pre>
<pre>size_t signature_length);</pre>

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 <pre>psa_algorithm_t</pre> such that <pre>PSA_ALG_IS_SIGN_HASH(alg)</pre> 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 <pre>psa_crypto_init().</pre>
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.

The following conditions can result in this error: PSA ERROR INVALID ARGUMENT • alg is not an asymmetric signature algorithm. • 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. The following conditions can result in this error: PSA\_ERROR\_NOT\_SUPPORTED • alg is not supported or is not an asymmetric signature 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

With most signature algorithms that follow the hash-and-sign paradigm, the hash input to this function is the hash of the message to verify. The algorithm used to compute this hash is encoded in the signature algorithm. For such algorithms, hash\_length must equal the length of the hash output, and the following condition is true:

hash\_length == PSA\_HASH\_LENGTH(PSA\_ALG\_GET\_HASH(alg))

The current version of this specification defines the following signature algorithms with this property: PSA\_ALG\_RSA\_PKCS1V15\_SIGN, PSA\_ALG\_RSA\_PSS, PSA\_ALG\_ECDSA, PSA\_ALG\_DETERMINISTIC\_ECDSA, PSA\_ALG\_ED25519PH, and PSA\_ALG\_ED448PH.

Some hash-and-sign mechanisms apply a padding or encoding to the hash. In such cases, the encoded hash must be passed to this function. The current version of this specification defines one such signature algorithm: PSA\_ALG\_RSA\_PKCS1V15\_SIGN\_RAW.

# Note:

To perform a hash-and-sign verification algorithm, the hash must be calculated before passing it to this function. This can be done by calling psa\_hash\_compute() or with a multi-part hash operation. The correct hash algorithm to use can be determined using PSA\_ALG\_GET\_HASH().

Alternatively, to hash and verify a message in a single call, use psa\_verify\_message().

# 10.7.3 Support macros

# PSA\_ALG\_IS\_SIGN\_MESSAGE (macro)

Whether the specified algorithm is a signature algorithm that can be used with psa\_sign\_message() and psa\_verify\_message().

#define PSA\_ALG\_IS\_SIGN\_MESSAGE(alg) /\* specification-defined value \*/

#### Parameters

alg

An algorithm identifier: 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 an already-calculated hash. 0 if alg is not a signature algorithm. This macro can return either 0 or 1 if alg is not a supported algorithm identifier.

# PSA\_ALG\_IS\_SIGN\_HASH (macro)

Whether the specified algorithm is a signature algorithm that can be used with psa\_sign\_hash() and psa\_verify\_hash().

#define PSA\_ALG\_IS\_SIGN\_HASH(alg) /\* specification-defined value \*/

#### Parameters

alg

An algorithm identifier: 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 includes all algorithms such that PSA\_ALG\_IS\_HASH\_AND\_SIGN() is true, as well as signature algorithms for which the input to psa\_sign\_hash() or psa\_verify\_hash() is not directly a hash, such as PSA\_ALG\_IS\_RSA\_PKCS1V15\_SIGN.

#### 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.

#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.

#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().

#### 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().

#### PSA\_ALG\_IS\_HASH\_EDDSA (macro)

Whether the specified algorithm is HashEdDSA.

#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.

# 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 parameterised 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, ...):

```
psa_hash_operation_t op = {0};
uint8_t hash[PSA_HASH_MAX_SIZE];
size_t hash_len;
psa_hash_setup(&op, PSA_ALG_GET_HASH(alg));
psa_hash_update(&op, msg, msg_len);
psa_hash_finish(&op, hash, sizeof(hash), &hash_len);
psa_sign_hash(key, alg, hash, hash_len, ...);
```

This excludes hash-and-sign algorithms that require a encoded or modified hash for the signature step in the algorithm, such as PSA\_ALG\_RSA\_PKCS1V15\_SIGN\_RAW. For such algorithms, PSA\_ALG\_IS\_SIGN\_HASH() is true but PSA\_ALG\_IS\_HASH\_AND\_SIGN() is false.

# 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().

# 10.8 Asymmetric encryption

Asymmetric encryption is provided through the functions psa\_asymmetric\_encrypt() and psa\_asymmetric\_decrypt().

# 10.8.1 Asymmetric encryption algorithms

#### PSA\_ALG\_RSA\_PKCS1V15\_CRYPT (macro)

The RSA PKCS#1 v1.5 asymmetric encryption algorithm.

#define PSA\_ALG\_RSA\_PKCS1V15\_CRYPT ((psa\_algorithm\_t)0x07000200)

This encryption scheme is defined by PKCS #1: RSA Cryptography Specifications Version 2.2 [RFC8017] §7.2 under the name RSAES-PKCS-v1\_5.

#### Compatible key types

PSA\_KEY\_TYPE\_RSA\_KEY\_PAIR PSA\_KEY\_TYPE\_RSA\_PUBLIC\_KEY (asymmetric encryption only)

#### PSA\_ALG\_RSA\_0AEP (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)
```

# 10.8.2 Asymmetric encryption functions

#### psa\_asymmetric\_encrypt (function)

Encrypt a short message with a public key.

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 <pre>psa_algorithm_t such that PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) is true.</pre>
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.
	<ul> <li>PSA_ASYMMETRIC_ENCRYPT_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported asymmetric encryption.</li> </ul>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not an asymmetric encryption algorithm.</li> <li>key is not a public key or an asymmetric key pair, that is compatible with alg.</li> </ul>
	<ul> <li>input_length is not valid for the algorithm and key type.</li> <li>salt_length is not valid for the algorithm and key type.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an asymmetric encryption algorithm.</li> </ul>
	<ul> <li>key is not supported for use with alg.</li> </ul>
	<ul> <li>input_length or salt_length are too large for the implementation.</li> </ul>
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_asymmetric_decrypt(psa_key_id_t k	ey,
psa_algorithm_	t alg,
const uint8_t	* input,
<pre>size_t input_l</pre>	ength,
const uint8_t	* salt,
size_t salt_le	ngth,
uint8_t * outp	ut,
<pre>size_t output_</pre>	size,
size_t * outpu	t_length)
p	<pre>sa_asymmetric_decrypt(psa_key_id_t ka</pre>

#### 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 <pre>psa_algorithm_t such that PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) is true.</pre>
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:
	<ul> <li>alg is not an asymmetric encryption algorithm.</li> <li>key is not an asymmetric key pair, that is compatible with alg.</li> <li>input_length is not valid for the algorithm and key type.</li> <li>salt_length is not valid for the algorithm and key type.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not an asymmetric encryption algorithm.</li> </ul>
	<ul> <li>key is not supported for use with alg.</li> </ul>
	<ul> <li>input_length or salt_length are too large for the implementation.</li> </ul>

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.

# 10.8.3 Support macros

# PSA\_ALG\_IS\_RSA\_0AEP (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().

# 10.9 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.

# 10.9.1 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 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  $Z = hd_AQ_B = hd_BQ_A$  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 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.

# 10.9.2 Standalone key agreement

# psa\_key\_agreement (function)

Perform a key agreement and return the shared secret as a derivation key.

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 in <i>Key formats</i> on page 105.
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. This function uses the attributes as follows:
	<ul> <li>The key type must be one of PSA_KEY_TYPE_DERIVE, PSA_KEY_TYPE_RAW_DATA, PSA_KEY_TYPE_HMAC, or PSA_KEY_TYPE_PASSWORD.</li> <li>Implementations must support the PSA_KEY_TYPE_DERIVE and PSA_KEY_TYPE_RAW_DATA key types.</li> </ul>
	<ul> <li>The size of the returned key is always the bit-size of the shared secret, rounded up to a whole number of bytes. The key size in attributes can be zero; if it is nonzero, it must be equal to the output size of the key agreement, in bits.</li> <li>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 bits are the type.     </li> </ul>
	<ul> <li>The key permitted-algorithm policy is required for keys that will be used for a cryptographic operation, see <i>Permitted algorithms</i> on page 85.</li> </ul>

	<ul> <li>The key usage flags define what operations are permitted with the key, see <i>Key usage flags</i> on page 87.</li> <li>The key lifetime and identifier are required for a persistent key.</li> </ul>
	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 <pre>psa_crypto_init().</pre>
PSA_ERROR_INVALID_HANDLE	private_key is not a valid key identifier.
PSA_ERROR_NOT_PERMITTED	The following conditions can result in this error:
	<ul> <li>private_key does not have the PSA_KEY_USAGE_DERIVE flag, or it does not permit the requested algorithm.</li> <li>The implementation does not permit creating a key with the specified attributes due to some implementation-specific policy.</li> </ul>
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:
	<ul> <li>alg is not a key agreement algorithm.</li> <li>private_key is not compatible with alg.</li> <li>peer_key is not a valid public key corresponding to private_key.</li> <li>The output key attributes in attributes are not valid : <ul> <li>The key type is not valid for key agreement output.</li> <li>The key size is nonzero, and is not the size of the shared secret.</li> <li>The key lifetime is invalid.</li> <li>The key identifier is not valid for the key lifetime.</li> <li>The key usage flags include invalid values.</li> <li>The key's permitted-usage algorithm is invalid.</li> </ul> </li> </ul>
PSA_ERROR_NOT_SUPPORTED	<ul> <li>The following conditions can result in this error:</li> <li>alg is not supported or is not a key agreement algorithm.</li> <li>private_key is not supported for use with alg.</li> <li>The output key attributes, as a whole, are not supported, either</li> </ul>

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. This key can be 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.

alg	The standalone key agreement algorithm to compute: a value of type <pre>psa_algorithm_t such that PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg) is true.</pre>
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 in <i>Key formats</i> on page 105.

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:
	<ul> <li>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. </li> <li>PSA_RAW_KEY_AGREEMENT_OUTPUT_MAX_SIZE evaluates to the maximum output size of any supported standalone key agreement algorithm.</li></ul>
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 <pre>psa_crypto_init().</pre>
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:
	<ul> <li>alg is not a key agreement algorithm.</li> <li>private_key is not compatible with alg.</li> <li>peer_key is not a valid public key corresponding to private_key.</li> </ul>
PSA_ERROR_NOT_SUPPORTED	The following conditions can result in this error:
	<ul> <li>alg is not supported or is not a key agreement algorithm.</li> <li>private_key is not supported for use with alg.</li> </ul>
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.

# 10.9.3 Combining key agreement and key derivation

#### psa\_key\_derivation\_key\_agreement (function)

Perform a key agreement and use the shared secret as input to a key derivation.

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 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 in <i>Key formats</i> on page 105.
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:
	<ul> <li>The operation state is not valid for this key agreement step.</li> <li>The library requires initializing by a call to psa_crypto_init().</li> </ul>
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:
	<ul> <li>The operation's algorithm is not a key agreement algorithm.</li> <li>step does not permit an input resulting from a key agreement.</li> <li>private_key is not compatible with the operation's algorithm.</li> <li>peer_key is not a valid public key corresponding to private_key.</li> </ul>
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.

# 10.9.4 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.

```
#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.

```
#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.

# Description

This is the original API name for PSA\_ALG\_IS\_STANDALONE\_KEY\_AGREEMENT().

# 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().

# 10.10 Other cryptographic services

# 10.10.1 Random number generation

# psa\_generate\_random (function)

Generate random bytes.

#### Parameters

	output	Output buffer for the generated data.	
	output_size	Number of bytes to generate and output.	
Re	Returns: psa_status_t		
	PSA_SUCCESS	Success. ${\tt output}\xspace$ output contains ${\tt output}\xspace$ bytes of generated random data.	
	PSA_ERROR_BAD_STATE	The library requires initializing by a call to <pre>psa_crypto_init().</pre>	
	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 key, use psa\_generate\_key() 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 v1.2
*/
#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 2
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 */
#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 */
#define PSA_KEY_TYPE_RAW_DATA ((psa_key_type_t)0x1001)
#define PSA_KEY_TYPE_HMAC ((psa_key_type_t)0x1100)
#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_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_RSA_KEY_PAIR ((psa_key_type_t)0x7001)
#define PSA_KEY_TYPE_RSA_PUBLIC_KEY ((psa_key_type_t)0x4001)
#define PSA_KEY_TYPE_IS_RSA(type) /* specification-defined value */
typedef uint8_t psa_ecc_family_t;
#define PSA_KEY_TYPE_ECC_KEY_PAIR(curve) /* specification-defined value */
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) /* specification-defined value */
#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)
#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 */
typedef uint8_t psa_dh_family_t;
#define PSA_KEY_TYPE_DH_KEY_PAIR(group) /* specification-defined value */
#define PSA_KEY_TYPE_DH_PUBLIC_KEY(group) /* specification-defined value */
#define PSA_DH_FAMILY_RFC7919 ((psa_dh_family_t) 0x03)
#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 */
```

```
#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 */
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);
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) 0x0000000)
#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))</pre>
#define PSA_KEY_ID_NULL ((psa_key_id_t)0)
#define PSA_KEY_ID_USER_MIN ((psa_key_id_t)0x0000001)
#define PSA_KEY_ID_USER_MAX ((psa_key_id_t)0x3fffffff)
#define PSA_KEY_ID_VENDOR_MIN ((psa_key_id_t)0x4000000)
#define PSA_KEY_ID_VENDOR_MAX ((psa_key_id_t)0x7ffffff)
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)0x0000002)
```

```
#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)
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_import_key(const psa_key_attributes_t * attributes,
                            const uint8_t * data,
                            size_t data_length,
                            psa_key_id_t * key);
psa_status_t psa_generate_key(const psa_key_attributes_t * attributes,
                              psa_key_id_t * 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);
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_ALG_NONE ((psa_algorithm_t)0)
#define PSA_ALG_IS_HASH(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_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_KEY_DERIVATION(alg) /* specification-defined value */
```
(continued from previous page)

```
#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)
#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)
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 */
#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_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);
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,
```

(continued from previous page)

```
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);
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);
```

(continued from previous page)

```
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 */
#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_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,
```

(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) \
```

(continued from previous page)

```
/* implementation-defined value */
#define PSA_AEAD_TAG_MAX_SIZE /* implementation-defined value */
#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_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_T0_MS(hash_alg) /* specification-defined value */
#define PSA_ALG_TLS12_ECJPAKE_TO_PMS ((psa_algorithm_t)0x08000609)
#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);
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\_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\_T0\_MS(alg) /\* specification-defined value \*/ #define PSA\_ALG\_IS\_PBKDF2\_HMAC(alg) /\* specification-defined value \*/ #define PSA\_KEY\_DERIVATION\_UNLIMITED\_CAPACITY \ /\* implementation-defined value \*/ #define PSA\_TLS12\_PSK\_T0\_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 \*/ #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\_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\_PURE\_EDDSA ((psa\_algorithm\_t) 0x06000800) #define PSA\_ALG\_ED25519PH ((psa\_algorithm\_t) 0x0600090B) #define PSA\_ALG\_ED448PH ((psa\_algorithm\_t) 0x06000915) 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\_verify\_message(psa\_key\_id\_t key, psa\_algorithm\_t alg, const uint8\_t \* input,

(continues on next page)

(continued from previous page)

```
size_t input_length,
                                const uint8_t * signature,
                                size_t signature_length);
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_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);
#define PSA_ALG_IS_SIGN_MESSAGE(alg) /* specification-defined value */
#define PSA_ALG_IS_SIGN_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_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_IS_HASH_EDDSA(alg) /* specification-defined value */
#define PSA_ALG_IS_HASH_AND_SIGN(alg) /* specification-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,
                                    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 */
#define PSA_ALG_IS_ECDH(alg) /* specification-defined value */
#define PSA_RAW_KEY_AGREEMENT_OUTPUT_SIZE(key_type, key_bits) \
    /* implementation-defined value */
```

(continued from previous page)

# 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 306 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 2.



Figure 2 Encoding of psa\_algorithm\_t

Table 10 on page 299 describes the meaning of the bit-fields — some of the bit-fields are used in different ways by different algorithm categories.

 Table 10 Bit fields in an algorithm identifier

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 derivation algorithm, this flag indicates a key-stretching or password-hashing algorithm when S=1.
В	[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.
Н	[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 11.

 Table 11 Algorithm identifier categories

Algorithm category	CAT	Category details
None	0x00	See PSA_ALG_NONE
Hash	0x02	See Hash algorithm encoding
MAC	0x03	See MAC algorithm encoding on page 300
Cipher	0x04	See Cipher algorithm encoding on page 301
AEAD	0x05	See AEAD algorithm encoding on page 302
Key derivation	0x08	See Key derivation algorithm encoding on page 303
Asymmetric signature	0x06	See Asymmetric signature algorithm encoding on page 304
Asymmetric encryption	0x07	See Asymmetric encryption algorithm encoding on page 305
Key agreement	0x09	See Key agreement algorithm encoding on page 305

### B.1.2 Hash algorithm encoding

The algorithm identifier for hash algorithms defined in this specification are encoded as shown in Figure 3.



Figure 3 Hash algorithm encoding

The defined values for HASH-TYPE are shown in Table 12 on page 300.

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
wildcard <sup>a</sup>	0xFF	PSA_ALG_ANY_HASH	0x020000FF

#### Table 12 Hash algorithm sub-type values

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 MAC algorithm encoding

The algorithm identifier for MAC algorithms defined in this specification are encoded as shown in Figure 4.

31	30		24	23	22	21		16	15	14				8	7							0
0		0x03		1	В		 LEN		W		MAC	-TYP	Έ			1	HAS	H-T	YP	Eor	0	

Figure 4 MAC algorithm encoding

The defined values for B and MAC-TYPE are shown in Table 13 on page 301.

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 12 on page 300) for hash-based MAC algorithms, otherwise H = 0.

 Table 13 MAC algorithm sub-type values

MAC algorithm	В	MAC-TYPE	Algorithm identifier	Algorithm value
HMAC	0	0×00	PSA_ALG_HMAC(hash_alg)	0x038000hh <sup>a b</sup>
CBC-MAC <sup>c</sup>	1	0x01	PSA_ALG_CBC_MAC	0x03c00100 <sup>a</sup>
CMAC <sup>c</sup>	1	0x02	PSA_ALG_CMAC	0x03c00200 <sup>a</sup>

- 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.1.4 Cipher algorithm encoding

The algorithm identifier for CIPHER algorithms defined in this specification are encoded as shown in Figure 5.



Figure 5 CIPHER algorithm encoding

The defined values for S, B, and CIPHER-TYPE are shown in Table 14 on page 302.

 Table 14 Cipher algorithm sub-type values

Cipher algorithm	S	В	CIPHER-TYPE	Algorithm identifier	Algorithm value
Stream cipher <sup>a</sup>	1	0	0x01	PSA_ALG_STREAM_CIPHER	0x04800100
CTR mode <sup>b</sup>	1	1	0x10	PSA_ALG_CTR	0x04C01000
CFB mode <sup>b</sup>	1	1	0x11	PSA_ALG_CFB	0x04C01100
OFB mode <sup>b</sup>	1	1	0x12	PSA_ALG_OFB	0x04C01200
$CCM^*$ with zero-length tag <sup>b</sup>	1	1	0x13	PSA_ALG_CCM_STAR_NO_TAG	0x04C01300
CCM <sup>*</sup> wildcard <sup>c</sup>	1	1	Øx93	PSA_ALG_CCM_STAR_ANY_TAG	0x04c09300
XTS mode <sup>b</sup>	0	1	0xFF	PSA_ALG_XTS	0x0440FF00
CBC mode without padding <sup>b</sup>	0	1	0x40	PSA_ALG_CBC_NO_PADDING	0x04404000
CBC mode with PKCS#7 padding $^{\rm b}$	0	1	0x41	PSA_ALG_CBC_PKCS7	0x04404100
ECB mode without padding <sup>b</sup>	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.1.5 AEAD algorithm encoding

The algorithm identifier for AEAD algorithms defined in this specification are encoded as shown in Figure 6.



Figure 6 AEAD algorithm encoding

The defined values for B and AEAD-TYPE are shown in Table 15 on page 303.

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 15 AEAD algorithm sub-type values

AEAD algorithm	В	AEAD-TYPE	Algorithm identifier	Algorithm value
CCM <sup>a</sup>	1	0x01	PSA_ALG_CCM	0x05500100 <sup>b</sup>
GCM <sup>a</sup>	1	0x02	PSA_ALG_GCM	0x05500200 <sup>b</sup>
ChaCha20-Poly1305	0	0x05	PSA_ALG_CHACHA20_POLY1305	0x05100500 <sup>b</sup>
XChaCha20-Poly1305	0	0x06	PSA_ALG_XCHACHA20_POLY1305	0x05100600 <sup>b</sup>

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.1.6 Key derivation algorithm encoding

The algorithm identifier for key derivation algorithms defined in this specification are encoded as shown in Figure 7.

31 3	30			24	23	22	21		16	15				8	-	7					0
0		0x08	1	1	s	0		0	1		КD	F-TY	′PE	1		1	HA	SH	-TY	PE	

Figure 7 Key derivation algorithm encoding

The defined values for S and KDF-TYPE are shown in Table 16.

The permitted values of HASH-TYPE (see Table 12 on page 300) depend on the specific KDF algorithm.

			·	
Key derivation algorithm	S	KDF- TYPE	Algorithm identifier	Algorithm value
HKDF	0	0x01	PSA_ALG_HKDF(hash)	0x080001hh <sup>a</sup>
TLS-1.2 PRF	0	0x02	PSA_ALG_TLS12_PRF (hash)	0x080002hh <sup>a</sup>
TLS-1.2 PSK-to-MasterSecret	0	0x03	PSA_ALG_TLS12_PSK_T0_MS(hash)	0x080003hh <sup>a</sup>
HKDF-Extract	0	0x04	PSA_ALG_HKDF_EXTRACT(hash)	0x080004hh <sup>a</sup>
HKDF-Expand	0	0x05	PSA_ALG_HKDF_EXPAND(hash)	0x080005hh <sup>a</sup>
TLS 1.2 ECJPAKE-to-PMS	0	0x06	PSA_ALG_TLS12_ECJPAKE_T0_PMS	0x08000609
SP 800-108 Counter HMAC	0	0x07	PSA_ALG_SP800_108_COUNTER_HMAC(hash)	0x080007hh <sup>a</sup>
SP 800-108 Counter CMAC	0	0x08	PSA_ALG_SP800_108_COUNTER_CMAC	0x08000800
PBKDF2-HMAC	1	0x01	PSA_ALG_PBKDF2_HMAC(hash)	0x088001hh <sup>a</sup>
PBKDF2-AES-CMAC-PRF- 128	1	0x02	PSA_ALG_PBKDF2_AES_CMAC_PRF_128	0x08800200

 Table 16 Key derivation algorithm sub-type values

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.

a. hh is the HASH-TYPE for the hash algorithm, hash, used to construct the key derivation algorithm.

## **B.1.7** Asymmetric signature algorithm encoding

The algorithm identifier for asymmetric signature algorithms defined in this specification are encoded as shown in Figure 8.

31	30	)					24	23	22	21					16	15							8	7						0
0				0x06	'	1		0	0		1		0	1	I		1	S	IGN	-TYI	PE	1	1			HAS	БН-1	' YPE	or 0	
		-	_									<b></b>	-	-	-			L	-	-	-	-			<u> </u>	<u> </u>	-	-	<u> </u>	

Figure 8 Asymmetric signature algorithm encoding

The defined values for SIGN-TYPE are shown in Table 17.

H = HASH-TYPE (see Table 12 on page 300) for message signature algorithms that are parameterized by a hash algorithm, otherwise H = 0.

Signature algorithm	SIGN-TYPE	Algorithm identifier	Algorithm value
RSA PKCS#1 v1.5	0x02	PSA_ALG_RSA_PKCS1V15_SIGN(hash_alg)	0x060002hh <sup>a</sup>
RSA PKCS#1 v1.5 no hash <sup>b</sup>	0x02	PSA_ALG_RSA_PKCS1V15_SIGN_RAW	0x06000200
RSA PSS	0x03	PSA_ALG_RSA_PSS(hash_alg)	0x060003hh <sup>a</sup>
RSA PSS any salt length	0x13	PSA_ALG_RSA_PSS_ANY_SALT(hash_alg)	0x060013hh <sup>a</sup>
Randomized ECDSA	0x06	PSA_ALG_ECDSA(hash_alg)	0x060006hh <sup>a</sup>
Randomized ECDSA no hash $^{\rm b}$	0x06	PSA_ALG_ECDSA_ANY	0x06000600
Deterministic ECDSA	0x07	PSA_ALG_DETERMINISTIC_ECDSA(hash_alg)	0x060007hh <sup>a</sup>
PureEdDSA	0x08	PSA_ALG_PURE_EDDSA	0x06000800
HashEdDSA	0x09	PSA_ALG_ED25519PH and PSA_ALG_ED448PH	0x060009hh <sup>c</sup>

 Table 17 Asymmetric signature algorithm sub-type values

- 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.1.8 Asymmetric encryption algorithm encoding

The algorithm identifier for asymmetric encryption algorithms defined in this specification are encoded as shown in Figure 9.

31	. 3	30				24	23	22	21			16	15						8	7							0
0				0x07			0	0			0			EN	ĊR١	/PT·	TYF	Έ			н.	ASł	-I-T	YPI	Eor	0	

Figure 9 Asymmetric encryption algorithm encoding

The defined values for ENCRYPT-TYPE are shown in Table 18.

H = HASH-TYPE (see Table 12 on page 300) for asymmetric encryption algorithms that are parameterized by a hash algorithm, otherwise H = 0.

		Table 18 Asymmetric encryption algorithm sub-typ							
Asymmetric encryption algorithm	ENCRYPT-TYPE	Algorithm identifier	Algorithm value						
RSA PKCS#1 v1.5	0x02	PSA_ALG_RSA_PKCS1V15_CRYPT	0x07000200						
RSA OAEP	0x03	<pre>PSA_ALG_RSA_0AEP(hash_alg)</pre>	0x070003hh <sup>a</sup>						

a. hh is the HASH-TYPE for the hash algorithm, hash\_alg, used to construct the encryption algorithm.

## B.1.9 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 10.

31	30		24	23	22	21				16	51	L5				8	7				0
0		0x09		0	0		к,	4-ΤY	Έ					0					0		

Figure 10 Standalone key agreement algorithm encoding

The defined values for KA-TYPE are shown in Table 19.

 Table 19 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 11 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.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 12.



Figure 12 Encoding of  $psa_key_type_t$ 

Table 20 describes the meaning of the bit-fields — some of bit-fields are used in different ways by different key type categories.

Table 20 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.
А	[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.
Ρ	[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 21.

				Table 21 Key type categories
Key type category	Α	CAT	Category details	
None	0	0	See PSA_KEY_TYPE_NONE	
Raw data	0	1	See Raw key encoding on page 307	
Symmetric key	0	2	See Symmetric key encoding on page 307	
Asymmetric public key	1	0	See Asymmetric key encoding on page 308	
Asymmetric key pair	1	3	See Asymmetric key encoding on page 308	

# B.2.2 Raw key encoding

The key type for raw keys defined in this specification are encoded as shown in Figure 13.

15	14	13 12	11 8	7 1	0
0	0	1	RAW-TYPE	SUB-TYPE	Р

Figure 13 Raw key encoding

The defined values for RAW-TYPE, SUB-TYPE, and P are shown in Table 22.

				Та	able 22 Raw key sub-type v
Raw key type	RAW-TYPE	SUB-TYPE	Ρ	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_HA	SH 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 14.

15 1	14 13	3 12	11	10	8	7				1	0
0	0	2	0	BLK			SYN	י א-ד`	YPE		Ρ

Figure 14 Symmetric key encoding

For block-based cipher keys, the block size for the cipher algorithm is 2<sup>BLK</sup>.

The defined values for BLK, SYM-TYPE and P are shown in Table 23 on page 308.

Table 23 Symmetric key sub-type values

Symmetric key type	BLK	SYM-TYPE	Ρ	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
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 Asymmetric key encoding

The key type for asymmetric keys defined in this specification are encoded as shown in Figure 15.

15 3	14	13 12	11	8	7		1	0
0	1	PAIR	ASY№	1-TYPE		FAMILY		Р

Figure 15 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 24.

Table 24 Asymmetric key sub-type values

Asymmetric key type	ASYM-TYPE	Details
RSA	0	See RSA key encoding
Elliptic Curve	1	See Elliptic Curve key encoding on page 309
Diffie-Hellman	2	See Diffie Hellman key encoding on page 310

#### **RSA** key encoding

The key type for RSA keys defined in this specification are encoded as shown in Figure 16.



#### Figure 16 RSA key encoding

PAIR is either 0 for a public key, or 3 for a key pair.

The defined values for RSA keys are shown in Table 25 on page 309.

Table 25 RSA key values

RSA key type	Key type	Key type value
Public key	PSA_KEY_TYPE_RSA_PUBLIC_KEY	0x4001
Key pair	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 17.



Figure 17 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 26.

				Table 26 EC	C key family values
ECC key family	ECC-FAMILY	Ρ	ECC family <sup>a</sup>	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 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.

#### Diffie Hellman key encoding

The key type for Diffie Hellman keys defined in this specification are encoded as shown in Figure 18.

15	14	13 12	11 8	7 1	0
0	1	PAIR	2	DH-FAMILY	Ρ

Figure 18 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 27.

 Table 27 Diffie Hellman key group values

DH key group	DH-FAMILY	Ρ	DH group <sup>a</sup>	Public key value	Key pair value
RFC7919	0x01	1	PSA_DH_FAMILY_RFC7919	0x4203	0x7203

a. The key type value is constructed from the Diffie Hellman family using either PSA\_KEY\_TYPE\_DH\_PUBLIC\_KEY(family) or PSA\_KEY\_TYPE\_DH\_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) & 0x00000ff))))
#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) & 0x00000ff))))
#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) & 0x00000ff))))
#define PSA_ALG_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x03800000 | ((hash_alg) & 0x00000ff)))
#define PSA_ALG_IS_AEAD(alg) \
    (((alg) & 0x7f000000) == 0x05000000)
#define PSA_ALG_IS_AEAD_ON_BLOCK_CIPHER(alg) \
    (((alg) & 0x7f400000) == 0x05400000)
```

```
#define PSA_ALG_IS_ASYMMETRIC_ENCRYPTION(alg) \
    (((alg) & 0x7f000000) == 0x07000000)
#define PSA_ALG_IS_BLOCK_CIPHER_MAC(alg) \
    (((alg) & 0x7fc00000) == 0x03c00000)
#define PSA_ALG_IS_CIPHER(alg) \
    (((alg) & 0x7f000000) == 0x04000000)
#define PSA_ALG_IS_DETERMINISTIC_ECDSA(alg) \
    (((alg) & ~0x00000ff) == 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) & ~0x00000ff) == 0x06000900)
#define PSA_ALG_IS_HKDF(alg) \
    (((alg) & ~0x00000ff) == 0x08000100)
#define PSA_ALG_IS_HKDF_EXPAND(alg) \
    (((alg) & ~0x00000ff) == 0x08000500)
#define PSA_ALG_IS_HKDF_EXTRACT(alg) \
    (((alg) & ~0x000000ff) == 0x08000400)
#define PSA_ALG_IS_HMAC(alg) \
    (((alg) & 0x7fc0ff00) == 0x03800000)
#define PSA_ALG_IS_KEY_AGREEMENT(alg) \
    (((alg) & 0x7f000000) == 0x09000000)
```

```
#define PSA_ALG_IS_KEY_DERIVATION(alg) \
    (((alg) & 0x7f000000) == 0x08000000)
#define PSA_ALG_IS_KEY_DERIVATION_STRETCHING(alg) \
    (((alg) & 0x7f800000) == 0x08800000)
#define PSA_ALG_IS_MAC(alg) \
    (((alg) & 0x7f000000) == 0x03000000)
#define PSA_ALG_IS_PBKDF2_HMAC(alg) \
    (((alg) & ~0x00000ff) == 0x08800100)
#define PSA_ALG_IS_RANDOMIZED_ECDSA(alg) \
    (((alg) & ~0x00000ff) == 0x06000600)
#define PSA_ALG_IS_RSA_OAEP(alg) \
    (((alg) & ~0x00000ff) == 0x07000300)
#define PSA_ALG_IS_RSA_PKCS1V15_SIGN(alg) \
    (((alg) & ~0x00000ff) == 0x06000200)
#define PSA_ALG_IS_RSA_PSS(alg) \
    (((alg) & ~0x000010ff) == 0x06000300)
#define PSA_ALG_IS_RSA_PSS_ANY_SALT(alg) \
    (((alg) & ~0x00000ff) == 0x06001300)
#define PSA_ALG_IS_RSA_PSS_STANDARD_SALT(alg) \
    (((alg) & ~0x00000ff) == 0x06000300)
#define PSA_ALG_IS_SIGN(alg) \
    (((alg) & 0x7f000000) == 0x06000000)
#define PSA_ALG_IS_SIGN_HASH(alg) \
   PSA_ALG_IS_SIGN(alg)
#define PSA_ALG_IS_SIGN_MESSAGE(alg) \
    (PSA_ALG_IS_SIGN(alg) && \
     (alg) != PSA_ALG_ECDSA_ANY && (alg) != PSA_ALG_RSA_PKCS1V15_SIGN_RAW)
#define PSA_ALG_IS_SP800_108_COUNTER_HMAC(alg) \
    (((alg) & ~0x00000ff) == 0x08000700)
#define PSA_ALG_IS_STANDALONE_KEY_AGREEMENT(alg) \
    (((alg) & 0x7f00ffff) == 0x0900000)
```

```
#define PSA_ALG_IS_STREAM_CIPHER(alg) \
    (((alg) & 0x7f800000) == 0x04800000)
#define PSA_ALG_IS_TLS12_PRF(alg) \
    (((alg) & ~0x00000ff) == 0x08000200)
#define PSA_ALG_IS_TLS12_PSK_TO_MS(alg) \
    (((alg) & ~0x00000ff) == 0x08000300)
#define PSA_ALG_IS_WILDCARD(alg) \
    ((PSA_ALG_GET_HASH(alg) == PSA_ALG_ANY_HASH) || \
     (((alg) & 0x7f008000) == 0x03008000) || \
     (((alg) & 0x7f008000) == 0x05008000))
#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) & 0x00000ff))))
#define PSA_ALG_RSA_PSS_ANY_SALT(hash_alg) \
    ((psa_algorithm_t)(0x06001300 | ((hash_alg) & 0x00000ff))))
#define PSA_ALG_SP800_108_COUNTER_HMAC(hash_alg) \
    ((psa_algorithm_t) (0x08000700 | ((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) & 0x00000ff))))
```

```
#define PSA_ALG_TRUNCATED_MAC(mac_alg, mac_length) \
    ((psa_algorithm_t) (((mac_alg) & ~0x003f8000) | (((mac_length) & 0x3f) << 16)))</pre>
```

# 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) & 0x00ff))
#define PSA_KEY_TYPE_DH_KEY_PAIR(group) \
    ((psa_key_type_t) (0x7200 | (group)))
#define PSA_KEY_TYPE_DH_PUBLIC_KEY(group) \
    ((psa_key_type_t) (0x4200 | (group)))
#define PSA_KEY_TYPE_ECC_GET_FAMILY(type) \
    ((psa_ecc_family_t) ((type) & 0x00ff))
#define PSA_KEY_TYPE_ECC_KEY_PAIR(curve) \
    ((psa_key_type_t) (0x7100 | (curve)))
#define PSA_KEY_TYPE_ECC_PUBLIC_KEY(curve) \
    ((psa_key_type_t) (0x4100 | (curve)))
#define PSA_KEY_TYPE_IS_ASYMMETRIC(type) \
    (((type) & 0x4000) == 0x4000)
#define PSA_KEY_TYPE_IS_DH(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff00) == 0x4200)
#define PSA_KEY_TYPE_IS_DH_KEY_PAIR(type) \
    (((type) & 0xff00) == 0x7200)
#define PSA_KEY_TYPE_IS_DH_PUBLIC_KEY(type) \
    (((type) & 0xff00) == 0x4200)
#define PSA_KEY_TYPE_IS_ECC(type) \
    ((PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) & 0xff00) == 0x4100)
#define PSA_KEY_TYPE_IS_ECC_KEY_PAIR(type) \
    (((type) & 0xff00) == 0x7100)
```

```
#define PSA_KEY_TYPE_IS_ECC_PUBLIC_KEY(type) \
    (((type) & 0xff00) == 0x4100)
#define PSA_KEY_TYPE_IS_KEY_PAIR(type) \
    (((type) & 0x7000) == 0x7000)
#define PSA_KEY_TYPE_IS_PUBLIC_KEY(type) \
    (((type) & 0x7000) == 0x4000)
#define PSA_KEY_TYPE_IS_RSA(type) \
    (PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) == 0x4001)
#define PSA_KEY_TYPE_IS_UNSTRUCTURED(type) \
    (((type) & 0x7000) == 0x1000 || ((type) & 0x7000) == 0x2000)
#define PSA_KEY_TYPE_KEY_PAIR_OF_PUBLIC_KEY(type) \
    ((psa_key_type_t) ((type) | 0x3000))
#define PSA_KEY_TYPE_PUBLIC_KEY_OF_KEY_PAIR(type) \
    ((psa_key_type_t) ((type) & ~0x3000))
```

# 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 + \
    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 320.

# **D.1** Architecture

### D.1.1 System definition

Figure 19 shows the Crypto API as the defined interface that an Application uses to interact with the Cryptoprocessor.



Figure 19 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 18.
- The API does not prevent the use of mitigations that are required by an implementation of the API. See *Implementation remediations* on page 329.
- 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 20.

#### 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 20 on page 318 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 20 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 320, and remediations R.1 and R.2 in *Implementation remediations* on page 329.

The Cryptoprocessor can optionally include a trust boundary within its implementation of the API. The trust boundary shown in Figure 21 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.



Figure 21 Crypto API dataflow diagram for an implementation with cryptoprocessor isolation

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 22 on page 319. The Application boundary restricts the capabilities of the Application, and protects the confidentiality and integrity of system state from the Application.



Figure 22 Crypto API dataflow diagram for an implementation with caller isolation

# 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.

- 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 28 lists the security goals for the Crypto API to address these threats.

Table 28 Security goals

ld	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 29 on page 321 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.

#### 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 30 on page 322.

Id	Description
M.3	The Adversary is capable of mounting sophisticated and active physical attacks. This includes, but is not limited to:
	<ul> <li>Interposing memory and blocking, replaying, and injecting transactions, this requires a much more precise timing than passive eavesdropping.</li> <li>Replacing or adding chips on the motherboard.</li> </ul>
M.4	The Adversary is capable of performing invasive silicon microsurgery.

### D.2.2 Threats and attacks

Table 31 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 324 for an evaluation of the risks posed by these threats, *Mitigations* on page 325 for mitigation requirements in the API design, and *Implementation remediations* on page 329 for mitigation recommendations in the cryptoprocessor implementation.

				Table 31 Threats and attacks
Threat				Attack (Examples)
ld	Description	Goals	Mod- els	Id: Description
T.1	Use of insecure or incorrectly implemented cryptography	G.1 G.2	M.0	<b>A.C1</b> : 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.
				<b>A.C2</b> : 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.
				<b>A.C3</b> : 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.

Threat				Attack (Examples)
ld	Description	Goals	Mod- els	Id: Description
Т.2	Misuse of cryptographic algorithms	G.1 G.2	M.0	<b>A.C4</b> : 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.
Т.З	Recover non-extractable key through the	G.3	M.1	<b>A.C5</b> : The attacker uses an indirect mechanism provided by the API to extract a key that is not intended to be extractable.
	API			<b>A.C6</b> : The attacker uses a mechanism provided by the API to enable brute-force recovery of a non-extractable key. For example, <i>On the Security of PKCS #11</i> [CLULOW] describes various flaws in the design of the PKCS #11 interface standard that enable an attacker to recover secret and non-extractable keys.
Т.4	Illegal inputs to the API	G.3 G.4	M.1	<b>A.60</b> : 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.
				<b>A.70</b> : Passing out-of-range values, or incorrectly formatted data, to provoke incorrect behavior in the cryptoprocessor.
				<b>A.61</b> : Providing invalid buffer lengths to cause out-of-bounds read or write access within the cryptoprocessor.
				<b>A.62</b> : 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	<b>A.C7</b> : Without a cryptoprocessor boundary, an attacker can directly access the cryptoprocessor state from an application. See Figure 20 on page 318.
				<b>A.C8</b> : A misconfigured cryptoprocessor boundary can allow an attacker to directly access the cryptoprocessor state from an Application.
Т.6	Access and use another application's assets	G.1 G.2	M.1	<b>A.C9</b> : 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 22 on page 319.
				within a caller-isolated implementation to gain access to another application's assets.

#### Table 31 - continued from previous page
Threat				Attack (Examples)
ld	Description	Goals	Mod- els	Id: Description
T.7	Data-dependent timing	G.1 G.3	M.1	<b>A.C11</b> 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.
T.8	Memory manipulation	G.4	M.2	<b>A.19</b> : Corrupt application or cryptoprocessor state via a fault, causing incorrect operation of the cryptoprocessor.
			M.1	<b>A.59</b> : Modifying function parameters in memory, while the cryptoprocessor is accessing the parameter memory, to cause incorrect operation of the cryptoprocessor.
Т.9	Side channels	G.1 G.3	M.2	<b>A.C12</b> 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	<b>A.C13</b> 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.

#### Table 31 - continued from previous page

#### D.2.3 Risk assessment

The risk ratings in Table 32 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 325 and *Implementation remediations* on page 329.

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 32 repeats the association in Table 31 on page 322 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.

Adversarial Model	Threat/Attack	Likelihood	Impact <sup>a</sup>	Risk
M.0	T.1	High	Medium	Medium
M.0	Т.2	High	Medium	Medium

#### continues on next page

Table 32 Risk assessment

Adversarial Model	Threat/Attack	Likelihood	Impact <sup>a</sup>	Risk
M.1	Т.3	Medium	High	Medium
M.1	T.4	High	Medium	Medium
M.1	Т.5	High	Very high	Very high
M.1	Т.6	High	High	High
M.1	Т.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 32 on page 324 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 33 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 326 and *Implementation remediations* on page 329.

		Table 33 Mitigation objectives
ld	Description	Threats addressed
0.1	Hide keys from the application	
	Keys are never directly manipulated by application software. Instead keys are referred to by handle,	T.1 T.2 T.3 — see A keystore interface on page 18.
	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.5 T.6 — to mitigate T.5 and T.6, the implementation must provide some form of isolation. See <i>Optional isolation</i> on page 18.

continues on next page

ld	Description	Threats addressed
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 <i>Key policies</i> on page 85.
O.3	Best-practice cryptography	
	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 <i>Ease of use</i> on page 20.
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 <i>Choice of algorithms</i> on page 19.

#### Table 33 - continued from previous page

#### D.3.2 Requirements

The design of the API can mitigate, or enable a cryptoprocessor to mitigate, some of the identified attacks. Table 34 describes these mitigations. Mitigations that are delegated to the cryptoprocessor or application are described in *Implementation remediations* on page 329.

Table 34 Security requirements

ld	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 <i>Key management</i> on page 21.	T.3/A.C5 — key values are hidden by the API.

continues on next page

Id	Description	API impact	Threats/attacks addressed
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 <i>Key policies</i> on page 85.	T.2/A.C4 — a key cannot be reused with different algorithms.
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().	<ul> <li>T.3/A.C5 – a copy of a key cannot be exported if the original could not be exported.</li> <li>T.3/A.C4 – a copy of a key cannot be used in different algorithm to the original.</li> </ul>
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 <i>Key types</i> on page 51 and <i>Algorithms</i> on page 108.	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

#### Table 34 - continued from previous page

ld	Description	API impact	Threats/attacks addressed
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 <i>Key type encoding</i> on page 306. <sup>a</sup> Success and error status codes differ by multiple bits, see <i>Status codes</i> on page 42. <sup>b</sup>	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.
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 <i>Overlap between parameters</i> on page 34.	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.

#### Table 34 - continued from previous page

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 35 includes all recommended remediations for an implementation, assuming the full adversarial model described in *Adversarial models* on page 320. 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.

Id	Identified gap	Suggested remediation
R.1 (O.1) (O.3)	T.5 – direct access to cryptoprocessor state.	The cryptoprocessor implementation provides <i>cryptoprocessor isolation</i> or <i>caller isolation</i> , 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 <i>caller isolation</i> , 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 <i>Multi-part operations</i> on page 24.
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 <i>Stability of parameters</i> on page 34.
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.

continues on next page

Table 35 Implementation remediations

Id	Identified gap	Suggested remediation
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.

#### Table 35 - continued from previous page

#### 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 325, or the cryptoprocessor implementation, as described in *Implementation remediations* on page 329.

Table 36 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.

		Idule 30 Residual TISK
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	Т.5	Correct isolation of the cryptoprocessor is the responsibility of the cryptoprocessor and system implementation.
RR.3	Т.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.

Table 21 Desidual viel

# E.1.1 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 310, 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.1.2 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.1.3 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. See *Key formats* on page 105.

#### E.1.4 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 viii.
- Adopted a standard set of Adversarial models for the Security Risk Assessment. See Adversarial models on page 320.

#### E.1.5 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 29.
- Added details for SHA-512/224 to the hash suspend state. See *Hash suspend state* on page 133.
- 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 298.
- Migrated cryptographic operation summaries to the start of the appropriate operation section, and out of the *Functionality overview* on page 21.
- Included a Security Risk Assessment for the Crypto API.

#### E.1.6 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 47 chapter.
  - Moved individual attribute types, values and accessor functions into their own sections.
  - Placed permitted algorithms and usage flags into *Key policies* on page 85.
  - Moved most introductory material from the *Functionality overview* on page 21 into the relevant API sections.

#### E.1.7 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\_T0\_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 37.
  - 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 310.
- 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\_T0\_MS\_MAX\_PSK\_LEN to PSA\_TLS12\_PSK\_T0\_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()
    - o PSA\_CIPHER\_DECRYPT\_OUTPUT\_SIZE()
    - o PSA\_CIPHER\_UPDATE\_OUTPUT\_SIZE()
    - PSA\_CIPHER\_FINISH\_OUTPUT\_SIZE()
    - PSA\_CIPHER\_IV\_LENGTH()
    - o PSA\_EXPORT\_PUBLIC\_KEY\_OUTPUT\_SIZE()
    - o PSA\_RAW\_KEY\_AGREEMENT\_OUTPUT\_SIZE()
  - Macros that evaluate to a maximum type-independent buffer size are provided. The new macros are:
    - o PSA\_AEAD\_ENCRYPT\_OUTPUT\_MAX\_SIZE()
    - o PSA\_AEAD\_DECRYPT\_OUTPUT\_MAX\_SIZE()
    - o 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()
    - o PSA\_CIPHER\_DECRYPT\_OUTPUT\_MAX\_SIZE()
    - o 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:
    - o PSA\_AEAD\_ENCRYPT\_OUTPUT\_SIZE()

- o PSA\_AEAD\_DECRYPT\_OUTPUT\_SIZE()
- PSA\_AEAD\_UPDATE\_OUTPUT\_SIZE()
- o 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:
  - $\circ \ \mathsf{PSA\_ALG\_AEAD\_WITH\_DEFAULT\_TAG\_LENGTH()} \to \mathsf{PSA\_ALG\_AEAD\_WITH\_DEFAULT\_LENGTH\_TAG()}$
  - $\circ \ \mathsf{PSA\_ALG\_AEAD\_WITH\_TAG\_LENGTH()} \rightarrow \mathsf{PSA\_ALG\_AEAD\_WITH\_SHORTENED\_TAG()}$
  - $\circ \ \mathsf{PSA\_KEY\_EXPORT\_MAX\_SIZE()} \rightarrow \mathsf{PSA\_EXPORT\_KEY\_OUTPUT\_SIZE()}$
  - $\circ$  PSA\_HASH\_SIZE()  $\rightarrow$  PSA\_HASH\_LENGTH()
  - $\circ$  PSA\_MAC\_FINAL\_SIZE()  $\rightarrow$  PSA\_MAC\_LENGTH()
  - $\circ$  PSA\_BLOCK\_CIPHER\_BLOCK\_SIZE()  $\rightarrow$  PSA\_BLOCK\_CIPHER\_BLOCK\_LENGTH()
  - $\circ \ \mathsf{PSA\_MAX\_BLOCK\_CIPHER\_BLOCK\_SIZE} \to \mathsf{PSA\_BLOCK\_CIPHER\_BLOCK\_MAX\_SIZE}$
- 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:
    - $\circ \ \mathsf{PSA\_KEY\_USAGE\_SIGN} \to \mathsf{PSA\_KEY\_USAGE\_SIGN\_HASH}$
    - $\circ \ \mathsf{PSA\_KEY\_USAGE\_VERIFY} \to \mathsf{PSA\_KEY\_USAGE\_VERIFY\_HASH}$
    - $\circ$  psa\_asymmetric\_sign()  $\rightarrow$  psa\_sign\_hash()
    - $\circ$  psa\_asymmetric\_verify()  $\rightarrow$  psa\_verify\_hash()
  - New APIs added to provide the complete message signing operation:
    - o PSA\_KEY\_USAGE\_SIGN\_MESSAGE
    - PSA\_KEY\_USAGE\_VERIFY\_MESSAGE
    - o psa\_sign\_message()
    - o psa\_verify\_message()
  - New Support macros to identify which algorithms can be used in which signing API:
    - PSA\_ALG\_IS\_SIGN\_HASH()
    - o PSA\_ALG\_IS\_SIGN\_MESSAGE()
  - Renamed support macros that apply to both signing APIs:
    - $\circ \ {\tt PSA\_ASYMMETRIC\_SIGN\_OUTPUT\_SIZE()} \to {\tt PSA\_SIGN\_OUTPUT\_SIZE()}$
    - $\circ \ \mathsf{PSA}\_\mathsf{ASYMMETRIC}\_\mathsf{SIGNATURE}\_\mathsf{MAX}\_\mathsf{SIZE} \to \mathsf{PSA}\_\mathsf{SIGNATURE}\_\mathsf{MAX}\_\mathsf{SIZE}$
  - 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
  - o PSA\_DH\_GROUP\_FFDHE3072
  - PSA\_DH\_GROUP\_FFDHE4096
  - PSA\_DH\_GROUP\_FFDHE6144
  - PSA\_DH\_GROUP\_FFDHE8192
  - o PSA\_ECC\_CURVE\_BITS
  - o PSA\_ECC\_CURVE\_BRAINPOOL\_P256R1
  - o PSA\_ECC\_CURVE\_BRAINPOOL\_P384R1
  - o PSA\_ECC\_CURVE\_BRAINPOOL\_P512R1
  - PSA\_ECC\_CURVE\_CURVE25519
  - o PSA\_ECC\_CURVE\_CURVE448
  - o PSA\_ECC\_CURVE\_SECP160K1
  - o PSA\_ECC\_CURVE\_SECP160R1
  - o 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
  - O 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
  - O PSA\_ECC\_CURVE\_SECT409K1
  - PSA\_ECC\_CURVE\_SECT409R1
  - PSA\_ECC\_CURVE\_SECT571K1
  - o 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
  - o PSA\_ECC\_FAMILY\_SECP\_R2
  - PSA\_ECC\_FAMILY\_SECT\_K1
  - o PSA\_ECC\_FAMILY\_SECT\_R1
  - PSA\_ECC\_FAMILY\_SECT\_R2
  - PSA\_ECC\_FAMILY\_MONTGOMERY
  - PSA\_KEY\_TYPE\_DH\_GET\_FAMILY
  - o PSA\_KEY\_TYPE\_ECC\_GET\_FAMILY
- The following macros have new values:
  - o PSA\_KEY\_TYPE\_AES
  - o PSA\_KEY\_TYPE\_ARC4
  - PSA\_KEY\_TYPE\_CAMELLIA
  - o 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
  - o 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
  - o 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
  - o PSA\_KEY\_TYPE\_IS\_DH\_PUBLIC\_KEY
  - PSA\_KEY\_TYPE\_IS\_ECC
  - PSA\_KEY\_TYPE\_IS\_ECC\_KEY\_PAIR
  - o PSA\_KEY\_TYPE\_IS\_ECC\_PUBLIC\_KEY
  - o PSA\_KEY\_TYPE\_IS\_KEY\_PAIR
  - o 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():
    - o PSA\_ALG\_HKDF\_GET\_HASH()
    - o PSA\_ALG\_HMAC\_GET\_HASH()
    - o PSA\_ALG\_RSA\_OAEP\_GET\_HASH()
    - o PSA\_ALG\_SIGN\_GET\_HASH()
    - PSA\_ALG\_TLS12\_PRF\_GET\_HASH()
    - o PSA\_ALG\_TLS12\_PSK\_T0\_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
  - o PSA\_ALG\_CBC\_PKCS7
  - PSA\_ALG\_CCM
  - PSA\_ALG\_CFB
  - o 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
  - o PSA\_ALG\_RSA\_PKCS1V15\_SIGN\_RAW
  - PSA\_ALG\_SHA\_1
  - PSA\_ALG\_SHA\_224
  - PSA\_ALG\_SHA\_256

- o PSA\_ALG\_SHA\_384
- o 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
- o PSA\_ALG\_SHA3\_512
- PSA\_ALG\_XTS
- The following macros with specification-defined values have new example implementations:
  - o PSA\_ALG\_AEAD\_WITH\_DEFAULT\_LENGTH\_TAG()
  - o PSA\_ALG\_AEAD\_WITH\_SHORTENED\_TAG()
  - o PSA\_ALG\_DETERMINISTIC\_ECDSA()
  - o PSA\_ALG\_ECDSA()
  - o PSA\_ALG\_FULL\_LENGTH\_MAC()
  - o PSA\_ALG\_HKDF()
  - o PSA\_ALG\_HMAC()
  - o PSA\_ALG\_IS\_AEAD()
  - o PSA\_ALG\_IS\_AEAD\_ON\_BLOCK\_CIPHER()
  - o PSA\_ALG\_IS\_ASYMMETRIC\_ENCRYPTION()
  - PSA\_ALG\_IS\_BLOCK\_CIPHER\_MAC()
  - PSA\_ALG\_IS\_CIPHER()
  - o PSA\_ALG\_IS\_DETERMINISTIC\_ECDSA()
  - o PSA\_ALG\_IS\_ECDH()
  - o PSA\_ALG\_IS\_ECDSA()
  - o PSA\_ALG\_IS\_FFDH()
  - o PSA\_ALG\_IS\_HASH()
  - o PSA\_ALG\_IS\_HASH\_AND\_SIGN()
  - o PSA\_ALG\_IS\_HKDF()
  - PSA\_ALG\_IS\_HMAC()
  - o PSA\_ALG\_IS\_KEY\_AGREEMENT()
  - o PSA\_ALG\_IS\_KEY\_DERIVATION()
  - o PSA\_ALG\_IS\_MAC()
  - o PSA\_ALG\_IS\_RANDOMIZED\_ECDSA()
  - o PSA\_ALG\_IS\_RAW\_KEY\_AGREEMENT()
  - o PSA\_ALG\_IS\_RSA\_OAEP()
  - o PSA\_ALG\_IS\_RSA\_PKCS1V15\_SIGN()
  - o PSA\_ALG\_IS\_RSA\_PSS()
  - o PSA\_ALG\_IS\_SIGN()
  - o PSA\_ALG\_IS\_SIGN\_MESSAGE()
  - o PSA\_ALG\_IS\_STREAM\_CIPHER()
  - o PSA\_ALG\_IS\_TLS12\_PRF()

- o PSA\_ALG\_IS\_TLS12\_PSK\_TO\_MS()
- PSA\_ALG\_IS\_WILDCARD()
- o PSA\_ALG\_KEY\_AGREEMENT()
- PSA\_ALG\_KEY\_AGREEMENT\_GET\_BASE()
- o PSA\_ALG\_KEY\_AGREEMENT\_GET\_KDF()
- o PSA\_ALG\_RSA\_OAEP()
- o PSA\_ALG\_RSA\_PKCS1V15\_SIGN()
- o PSA\_ALG\_RSA\_PSS()
- o PSA\_ALG\_TLS12\_PRF()
- PSA\_ALG\_TLS12\_PSK\_TO\_MS()
- o 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 133, 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.1.8 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  $\rightarrow$  PSA\_ERROR\_GENERIC\_ERROR
  - PSA\_ERROR\_OCCUPIED\_SLOT  $\rightarrow$  PSA\_ERROR\_ALREADY\_EXISTS
  - PSA\_ERROR\_EMPTY\_SLOT  $\rightarrow$  PSA\_ERROR\_DOES\_NOT\_EXIST
  - PSA\_ERROR\_INSUFFICIENT\_CAPACITY  $\rightarrow$  PSA\_ERROR\_INSUFFICIENT\_DATA
  - PSA\_ERROR\_TAMPERING\_DETECTED  $\rightarrow$  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  $\rightarrow$  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  $\rightarrow$  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()  $\rightarrow$  psa\_key\_derivation\_output\_key()
- psa\_generator\_abort() → psa\_key\_derivation\_abort()
- psa\_key\_agreement\_raw\_shared\_secret()  $\rightarrow$  psa\_raw\_key\_agreement()
- PSA\_KDF\_STEP\_xxx  $\rightarrow$  PSA\_KEY\_DERIVATION\_INPUT\_xxx
- PSA\_xxx\_KEYPAIR  $\rightarrow$  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.1.9 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 1.2.x

Future versions of this specification that use a 1.2.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:

- Multi-part operations for hybrid cryptography. For example, this includes hash-and-sign for EdDSA, and hybrid encryption for ECIES.
- Key wrapping mechanisms to extract and import keys in an encrypted and authenticated form.
- Key discovery mechanisms. This would enable an application to locate a key by its name or attributes.
- Implementation capability description. This would enable an application to determine the algorithms, key types and storage lifetimes that the implementation provides.
- An ownership and access control mechanism allowing a multi-client implementation to have privileged clients that are able to manage keys of other clients.

# **Index of API elements**

#### PSA\_A

PSA\_AEAD\_DECRYPT\_OUTPUT\_MAX\_SIZE, 205 PSA\_AEAD\_DECRYPT\_OUTPUT\_SIZE, 204 PSA\_AEAD\_ENCRYPT\_OUTPUT\_MAX\_SIZE, 204 PSA\_AEAD\_ENCRYPT\_OUTPUT\_SIZE, 203 PSA\_AEAD\_FINISH\_OUTPUT\_MAX\_SIZE, 207 PSA\_AEAD\_FINISH\_OUTPUT\_SIZE, 207 PSA\_AEAD\_NONCE\_LENGTH, 205 PSA\_AEAD\_NONCE\_MAX\_SIZE, 206 PSA\_AEAD\_OPERATION\_INIT, 189 PSA\_AEAD\_TAG\_LENGTH, 208 PSA\_AEAD\_TAG\_MAX\_SIZE, 208 PSA\_AEAD\_UPDATE\_OUTPUT\_MAX\_SIZE, 207 PSA\_AEAD\_UPDATE\_OUTPUT\_SIZE, 206 PSA\_AEAD\_VERIFY\_OUTPUT\_MAX\_SIZE, 209 PSA\_AEAD\_VERIFY\_OUTPUT\_SIZE, 208 PSA\_ALG\_AEAD\_WITH\_AT\_LEAST\_THIS\_LENGTH\_TAG, 183 PSA\_ALG\_AEAD\_WITH\_DEFAULT\_LENGTH\_TAG, 182 PSA\_ALG\_AEAD\_WITH\_SHORTENED\_TAG, 182 PSA\_ALG\_AES\_MMO\_ZIGBEE, 116 PSA\_ALG\_ANY\_HASH, 262 PSA\_ALG\_AT\_LEAST\_THIS\_LENGTH\_MAC, 139 PSA\_ALG\_CBC\_MAC, 136 PSA\_ALG\_CBC\_NO\_PADDING, 158 PSA\_ALG\_CBC\_PKCS7, 159 PSA\_ALG\_CCM, 179 PSA\_ALG\_CCM\_STAR\_ANY\_TAG, 173 PSA\_ALG\_CCM\_STAR\_NO\_TAG, 155 PSA\_ALG\_CFB, 156 PSA\_ALG\_CHACHA20\_POLY1305, 181 PSA\_ALG\_CMAC, 137 PSA\_ALG\_CTR, 153 PSA\_ALG\_DETERMINISTIC\_ECDSA, 247 PSA\_ALG\_ECB\_NO\_PADDING, 157 PSA\_ALG\_ECDH, 272 PSA\_ALG\_ECDSA, 245 PSA\_ALG\_ECDSA\_ANY, 247 PSA\_ALG\_ED25519PH, 249 PSA\_ALG\_ED448PH, 250 PSA\_ALG\_FFDH, 271

PSA\_ALG\_FULL\_LENGTH\_MAC, 138 PSA\_ALG\_GCM, 180 PSA\_ALG\_GET\_HASH, 113 PSA\_ALG\_HKDF, 210 PSA ALG HKDF EXPAND, 212 PSA\_ALG\_HKDF\_EXTRACT, 211 PSA\_ALG\_HMAC, 136 PSA\_ALG\_IS\_AEAD, 111 PSA\_ALG\_IS\_AEAD\_ON\_BLOCK\_CIPHER, 203 PSA\_ALG\_IS\_ASYMMETRIC\_ENCRYPTION, 112 PSA\_ALG\_IS\_BLOCK\_CIPHER\_MAC, 150 PSA\_ALG\_IS\_CIPHER, 111 PSA\_ALG\_IS\_DETERMINISTIC\_ECDSA, 260 PSA\_ALG\_IS\_ECDH, 282 PSA\_ALG\_IS\_ECDSA, 260 PSA\_ALG\_IS\_FFDH, 281 PSA\_ALG\_IS\_HASH, 110 PSA\_ALG\_IS\_HASH\_AND\_SIGN, 261 PSA\_ALG\_IS\_HASH\_EDDSA, 261 PSA\_ALG\_IS\_HKDF, 239 PSA\_ALG\_IS\_HKDF\_EXPAND, 240 PSA\_ALG\_IS\_HKDF\_EXTRACT, 239 PSA\_ALG\_IS\_HMAC, 149 PSA\_ALG\_IS\_KEY\_AGREEMENT, 112 PSA\_ALG\_IS\_KEY\_DERIVATION, 113 PSA\_ALG\_IS\_KEY\_DERIVATION\_STRETCHING, 238 PSA\_ALG\_IS\_MAC, 110 PSA\_ALG\_IS\_PBKDF2\_HMAC, 241 PSA\_ALG\_IS\_RANDOMIZED\_ECDSA, 261 PSA\_ALG\_IS\_RAW\_KEY\_AGREEMENT, 281 PSA\_ALG\_IS\_RSA\_OAEP, 269 PSA\_ALG\_IS\_RSA\_PKCS1V15\_SIGN, 258 PSA\_ALG\_IS\_RSA\_PSS, 259 PSA\_ALG\_IS\_RSA\_PSS\_ANY\_SALT, 259 PSA\_ALG\_IS\_RSA\_PSS\_STANDARD\_SALT, 260 PSA\_ALG\_IS\_SIGN, 111 PSA\_ALG\_IS\_SIGN\_HASH, 258 PSA\_ALG\_IS\_SIGN\_MESSAGE, 258 PSA\_ALG\_IS\_SP800\_108\_COUNTER\_HMAC, 240 PSA\_ALG\_IS\_STANDALONE\_KEY\_AGREEMENT, 280

PSA\_ALG\_IS\_STREAM\_CIPHER, 172 PSA\_ALG\_IS\_TLS12\_PRF, 240 PSA\_ALG\_IS\_TLS12\_PSK\_TO\_MS, 241 PSA\_ALG\_IS\_WILDCARD, 113 PSA\_ALG\_KEY\_AGREEMENT, 273 PSA\_ALG\_KEY\_AGREEMENT\_GET\_BASE, 280 PSA\_ALG\_KEY\_AGREEMENT\_GET\_KDF, 280 PSA\_ALG\_MD2, 115 PSA\_ALG\_MD4, 115 PSA\_ALG\_MD5, 115 PSA\_ALG\_NONE, 110 PSA\_ALG\_OFB, 156 PSA\_ALG\_PBKDF2\_AES\_CMAC\_PRF\_128, 219 PSA\_ALG\_PBKDF2\_HMAC, 218 PSA\_ALG\_PURE\_EDDSA, 248 PSA\_ALG\_RIPEMD160, 116 PSA\_ALG\_RSA\_OAEP, 264 PSA\_ALG\_RSA\_PKCS1V15\_CRYPT, 264 PSA\_ALG\_RSA\_PKCS1V15\_SIGN, 242 PSA\_ALG\_RSA\_PKCS1V15\_SIGN\_RAW, 243 PSA\_ALG\_RSA\_PSS, 243 PSA\_ALG\_RSA\_PSS\_ANY\_SALT, 244 PSA\_ALG\_SHA3\_224, 117 PSA\_ALG\_SHA3\_256, 118 PSA\_ALG\_SHA3\_384, 118 PSA\_ALG\_SHA3\_512, 118 PSA\_ALG\_SHAKE256\_512, 118 PSA\_ALG\_SHA\_1, 116 PSA\_ALG\_SHA\_224, 116 PSA\_ALG\_SHA\_256, 117 PSA\_ALG\_SHA\_384, 117 PSA\_ALG\_SHA\_512, 117 PSA\_ALG\_SHA\_512\_224, 117 PSA\_ALG\_SHA\_512\_256, 117 PSA\_ALG\_SM3, 118 PSA\_ALG\_SP800\_108\_COUNTER\_CMAC, 214 PSA\_ALG\_SP800\_108\_COUNTER\_HMAC, 213 PSA\_ALG\_STREAM\_CIPHER, 152 PSA\_ALG\_TLS12\_ECJPAKE\_TO\_PMS, 218 PSA\_ALG\_TLS12\_PRF, 215 PSA\_ALG\_TLS12\_PSK\_TO\_MS, 216 PSA\_ALG\_TRUNCATED\_MAC, 137 PSA\_ALG\_XCHACHA20\_POLY1305, 181 PSA\_ALG\_XTS, 157 PSA\_ASYMMETRIC\_DECRYPT\_OUTPUT\_MAX\_SIZE, 270 PSA\_ASYMMETRIC\_DECRYPT\_OUTPUT\_SIZE, 270 PSA\_ASYMMETRIC\_ENCRYPT\_OUTPUT\_MAX\_SIZE, 270 PSA\_ASYMMETRIC\_ENCRYPT\_OUTPUT\_SIZE, 269

psa\_aead\_abort, 202 psa\_aead\_decrypt, 186 psa\_aead\_decrypt\_setup, 191 psa\_aead\_encrypt, 184 psa\_aead\_encrypt\_setup, 189 psa\_aead\_finish, 199 psa\_aead\_generate\_nonce, 193 psa\_aead\_operation\_init, 189 psa\_aead\_operation\_t, 188 psa\_aead\_set\_lengths, 192 psa\_aead\_set\_nonce, 194 psa\_aead\_update, 197 psa\_aead\_update\_ad, 196 psa\_aead\_verify, 201 psa\_algorithm\_t, 109 psa\_asymmetric\_decrypt, 267 psa\_asymmetric\_encrypt, 265

# PSA\_B

PSA\_BLOCK\_CIPHER\_BLOCK\_LENGTH, 178 PSA\_BLOCK\_CIPHER\_BLOCK\_MAX\_SIZE, 178

# PSA\_C

PSA\_CIPHER\_DECRYPT\_OUTPUT\_MAX\_SIZE, 175 PSA\_CIPHER\_DECRYPT\_OUTPUT\_SIZE, 174 PSA\_CIPHER\_ENCRYPT\_OUTPUT\_MAX\_SIZE, 174 PSA\_CIPHER\_ENCRYPT\_OUTPUT\_SIZE, 173 PSA\_CIPHER\_FINISH\_OUTPUT\_MAX\_SIZE, 177 PSA\_CIPHER\_FINISH\_OUTPUT\_SIZE, 177 PSA\_CIPHER\_IV\_LENGTH, 175 PSA\_CIPHER\_IV\_MAX\_SIZE, 176 PSA\_CIPHER\_OPERATION\_INIT, 163 PSA\_CIPHER\_UPDATE\_OUTPUT\_MAX\_SIZE, 177 PSA\_CIPHER\_UPDATE\_OUTPUT\_SIZE, 176 PSA\_CRYPTO\_API\_VERSION\_MAJOR, 45 PSA\_CRYPTO\_API\_VERSION\_MINOR, 45 psa\_cipher\_abort, 172 psa\_cipher\_decrypt, 161 psa\_cipher\_decrypt\_setup, 165 psa\_cipher\_encrypt, 159 psa\_cipher\_encrypt\_setup, 164 psa\_cipher\_finish, 171 psa\_cipher\_generate\_iv, 167 psa\_cipher\_operation\_init, 163 psa\_cipher\_operation\_t, 163 psa\_cipher\_set\_iv, 168 psa\_cipher\_update, 169 psa\_copy\_key, 96

psa\_crypto\_init, 45

#### PSA\_D

PSA\_DH\_FAMILY\_RFC7919, 70
psa\_destroy\_key, 98
psa\_dh\_family\_t, 69

# PSA\_E

PSA\_ECC\_FAMILY\_BRAINPOOL\_P\_R1, 66 PSA\_ECC\_FAMILY\_FRP, 67 PSA\_ECC\_FAMILY\_MONTGOMERY, 67 PSA\_ECC\_FAMILY\_SECP\_K1, 63 PSA\_ECC\_FAMILY\_SECP\_R1, 64 PSA\_ECC\_FAMILY\_SECP\_R2, 64 PSA\_ECC\_FAMILY\_SECT\_K1, 65 PSA\_ECC\_FAMILY\_SECT\_R1, 65 PSA\_ECC\_FAMILY\_SECT\_R2, 66 PSA\_ECC\_FAMILY\_TWISTED\_EDWARDS, 68 PSA\_ERROR\_INSUFFICIENT\_ENTROPY, 44 PSA\_ERROR\_INVALID\_PADDING, 45 PSA\_EXPORT\_KEY\_OUTPUT\_SIZE, 103 PSA\_EXPORT\_KEY\_PAIR\_MAX\_SIZE, 105 PSA\_EXPORT\_PUBLIC\_KEY\_MAX\_SIZE, 105 PSA\_EXPORT\_PUBLIC\_KEY\_OUTPUT\_SIZE, 104 psa\_ecc\_family\_t, 62 psa\_export\_key, 100 psa\_export\_public\_key, 102

# PSA\_G

psa\_generate\_key, 95
psa\_generate\_random, 283
psa\_get\_key\_algorithm, 86
psa\_get\_key\_attributes, 50
psa\_get\_key\_bits, 73
psa\_get\_key\_id, 85
psa\_get\_key\_lifetime, 81
psa\_get\_key\_type, 73
psa\_get\_key\_usage\_flags, 92

# PSA\_H

PSA\_HASH\_BLOCK\_LENGTH, 132 PSA\_HASH\_LENGTH, 130 PSA\_HASH\_MAX\_SIZE, 130 PSA\_HASH\_OPERATION\_INIT, 121 PSA\_HASH\_SUSPEND\_ALGORITHM\_FIELD\_LENGTH, 131 PSA\_HASH\_SUSPEND\_HASH\_STATE\_FIELD\_LENGTH, 132 PSA\_HASH\_SUSPEND\_INPUT\_LENGTH\_FIELD\_LENGTH, 131 PSA\_HASH\_SUSPEND\_OUTPUT\_MAX\_SIZE, 131 PSA\_HASH\_SUSPEND\_OUTPUT\_SIZE, 130 psa\_hash\_abort, 126 psa\_hash\_clone, 129 psa\_hash\_compare, 120 psa\_hash\_compute, 119 psa\_hash\_finish, 123 psa\_hash\_operation\_init, 121 psa\_hash\_operation\_t, 121 psa\_hash\_resume, 128 psa\_hash\_resume, 128 psa\_hash\_setup, 122 psa\_hash\_suspend, 126 psa\_hash\_suspend, 123 psa\_hash\_verify, 125

# PSA\_I

psa\_import\_key, 93

# PSA\_K

PSA\_KEY\_ATTRIBUTES\_INIT, 50 PSA\_KEY\_DERIVATION\_INPUT\_CONTEXT, 221 PSA\_KEY\_DERIVATION\_INPUT\_COST, 221 PSA\_KEY\_DERIVATION\_INPUT\_INFO, 221 PSA\_KEY\_DERIVATION\_INPUT\_LABEL, 221 PSA\_KEY\_DERIVATION\_INPUT\_OTHER\_SECRET, 220 PSA\_KEY\_DERIVATION\_INPUT\_PASSWORD, 220 PSA\_KEY\_DERIVATION\_INPUT\_SALT, 221 PSA\_KEY\_DERIVATION\_INPUT\_SECRET, 220 PSA\_KEY\_DERIVATION\_INPUT\_SEED, 221 PSA\_KEY\_DERIVATION\_OPERATION\_INIT, 222 PSA\_KEY\_DERIVATION\_UNLIMITED\_CAPACITY, 241 PSA\_KEY\_ID\_NULL, 83 PSA\_KEY\_ID\_USER\_MAX, 83 PSA\_KEY\_ID\_USER\_MIN, 83 PSA\_KEY\_ID\_VENDOR\_MAX, 84 PSA\_KEY\_ID\_VENDOR\_MIN, 84 PSA\_KEY\_LIFETIME\_FROM\_PERSISTENCE\_AND\_LOCATION, 82 PSA\_KEY\_LIFETIME\_GET\_LOCATION, 81 PSA\_KEY\_LIFETIME\_GET\_PERSISTENCE, 81 PSA\_KEY\_LIFETIME\_IS\_VOLATILE, 82 PSA\_KEY\_LIFETIME\_PERSISTENT, 79 PSA\_KEY\_LIFETIME\_VOLATILE, 78 PSA\_KEY\_LOCATION\_LOCAL\_STORAGE, 79 PSA\_KEY\_LOCATION\_PRIMARY\_SECURE\_ELEMENT, 80 PSA\_KEY\_PERSISTENCE\_DEFAULT, 79 PSA\_KEY\_PERSISTENCE\_READ\_ONLY, 79 PSA\_KEY\_PERSISTENCE\_VOLATILE, 79 PSA\_KEY\_TYPE\_AES, 56 PSA\_KEY\_TYPE\_ARC4, 60

PSA\_KEY\_TYPE\_ARIA, 57 PSA\_KEY\_TYPE\_CAMELLIA, 58 PSA\_KEY\_TYPE\_CHACHA20, 60 PSA\_KEY\_TYPE\_DERIVE, 54 PSA\_KEY\_TYPE\_DES, 57 PSA\_KEY\_TYPE\_DH\_GET\_FAMILY, 72 PSA\_KEY\_TYPE\_DH\_KEY\_PAIR, 70 PSA\_KEY\_TYPE\_DH\_PUBLIC\_KEY, 70 PSA\_KEY\_TYPE\_ECC\_GET\_FAMILY, 69 PSA\_KEY\_TYPE\_ECC\_KEY\_PAIR, 62 PSA\_KEY\_TYPE\_ECC\_PUBLIC\_KEY, 63 PSA\_KEY\_TYPE\_HMAC, 53 PSA\_KEY\_TYPE\_IS\_ASYMMETRIC, 52 PSA\_KEY\_TYPE\_IS\_DH, 71 PSA\_KEY\_TYPE\_IS\_DH\_KEY\_PAIR, 71 PSA\_KEY\_TYPE\_IS\_DH\_PUBLIC\_KEY, 72 PSA\_KEY\_TYPE\_IS\_ECC, 68 PSA\_KEY\_TYPE\_IS\_ECC\_KEY\_PAIR, 68 PSA\_KEY\_TYPE\_IS\_ECC\_PUBLIC\_KEY, 69 PSA\_KEY\_TYPE\_IS\_KEY\_PAIR, 53 PSA\_KEY\_TYPE\_IS\_PUBLIC\_KEY, 52 PSA\_KEY\_TYPE\_IS\_RSA, 62 PSA\_KEY\_TYPE\_IS\_UNSTRUCTURED, 52 PSA\_KEY\_TYPE\_KEY\_PAIR\_OF\_PUBLIC\_KEY, 70 PSA\_KEY\_TYPE\_NONE, 52 PSA\_KEY\_TYPE\_PASSWORD, 55 PSA\_KEY\_TYPE\_PASSWORD\_HASH, 55 PSA\_KEY\_TYPE\_PEPPER, 55 PSA\_KEY\_TYPE\_PUBLIC\_KEY\_OF\_KEY\_PAIR, 71 PSA\_KEY\_TYPE\_RAW\_DATA, 53 PSA\_KEY\_TYPE\_RSA\_KEY\_PAIR, 61 PSA\_KEY\_TYPE\_RSA\_PUBLIC\_KEY, 61 PSA\_KEY\_TYPE\_SM4, 59 PSA\_KEY\_TYPE\_XCHACHA20, 60 PSA\_KEY\_USAGE\_CACHE, 88 PSA\_KEY\_USAGE\_COPY, 88 PSA\_KEY\_USAGE\_DECRYPT, 89 PSA\_KEY\_USAGE\_DERIVE, 91 PSA\_KEY\_USAGE\_ENCRYPT, 88 PSA\_KEY\_USAGE\_EXPORT, 87 PSA\_KEY\_USAGE\_SIGN\_HASH, 90 PSA\_KEY\_USAGE\_SIGN\_MESSAGE, 89 PSA\_KEY\_USAGE\_VERIFY\_DERIVATION, 91 PSA\_KEY\_USAGE\_VERIFY\_HASH, 90 PSA\_KEY\_USAGE\_VERIFY\_MESSAGE, 90 psa\_key\_agreement, 274 psa\_key\_attributes\_init, 50 psa\_key\_attributes\_t, 47

psa\_key\_derivation\_abort, 238 psa\_key\_derivation\_get\_capacity, 224 psa\_key\_derivation\_input\_bytes, 226 psa\_key\_derivation\_input\_integer, 227 psa\_key\_derivation\_input\_key, 228 psa\_key\_derivation\_key\_agreement, 278 psa\_key\_derivation\_operation\_init, 222 psa\_key\_derivation\_operation\_t, 222 psa\_key\_derivation\_output\_bytes, 230 psa\_key\_derivation\_output\_key, 231 psa\_key\_derivation\_set\_capacity, 225 psa\_key\_derivation\_setup, 223 psa\_key\_derivation\_step\_t, 220 psa\_key\_derivation\_verify\_bytes, 235 psa\_key\_derivation\_verify\_key, 236 psa\_key\_id\_t, 83 psa\_key\_lifetime\_t, 75 psa\_key\_location\_t, 77 psa\_key\_persistence\_t, 76 psa\_key\_type\_t, 51 psa\_key\_usage\_t, 87

#### PSA\_M

PSA\_MAC\_LENGTH, 150 PSA\_MAC\_MAX\_SIZE, 151 PSA\_MAC\_OPERATION\_INIT, 143 psa\_mac\_abort, 149 psa\_mac\_compute, 139 psa\_mac\_operation\_init, 143 psa\_mac\_operation\_t, 142 psa\_mac\_sign\_finish, 147 psa\_mac\_sign\_setup, 143 psa\_mac\_update, 146 psa\_mac\_verify, 141 psa\_mac\_verify\_finish, 148 psa\_mac\_verify\_setup, 144

# PSA\_P

psa\_purge\_key, 100

# PSA\_R

PSA\_RAW\_KEY\_AGREEMENT\_OUTPUT\_MAX\_SIZE, 282 PSA\_RAW\_KEY\_AGREEMENT\_OUTPUT\_SIZE, 282 psa\_raw\_key\_agreement, 276 psa\_reset\_key\_attributes, 51

# PSA\_S

PSA\_SIGNATURE\_MAX\_SIZE, 264

PSA\_SIGN\_OUTPUT\_SIZE, 263 psa\_set\_key\_algorithm, 86 psa\_set\_key\_bits, 73 psa\_set\_key\_id, 84 psa\_set\_key\_lifetime, 80 psa\_set\_key\_type, 72 psa\_set\_key\_usage\_flags, 91 psa\_sign\_hash, 254 psa\_sign\_message, 251

# PSA\_T

PSA\_TLS12\_ECJPAKE\_T0\_PMS\_OUTPUT\_SIZE, 242 PSA\_TLS12\_PSK\_T0\_MS\_PSK\_MAX\_SIZE, 241

#### PSA\_V

psa\_verify\_hash, 256
psa\_verify\_message, 253