



PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0

Committee Specification Draft 01 / Public Review Draft 01

29 May 2019

This version:

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.docx>

(Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.pdf>

Previous version:

N/A

Latest version:

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.docx> (Authoritative)

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html>

<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.pdf>

Technical Committee:

OASIS PKCS 11 TC

Chairs:

Tony Cox (tony.cox@cryptsoft.com), Cryptsoft Pty Ltd

Robert Relyea (rrelyea@redhat.com), Red Hat

Editors:

Chris Zimman (chris@wmpp.com), Individual

Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- PKCS #11 header files:
<https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/include/pkcs11-v3.0/>

Related work:

This specification replaces or supersedes:

- *PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 2.40*. Edited by Susan Gleeson, Chris Zimman, Robert Griffin, and Tim Hudson. Latest version. <http://docs.oasis-open.org/pkcs11/pkcs11-curr/v2.40/pkcs11-curr-v2.40.html>.

This specification is related to:

- *PKCS #11 Cryptographic Token Interface Profiles Version 3.0*. Edited by Tim Hudson. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html>.
- *PKCS #11 Cryptographic Token Interface Base Specification Version 3.0*. Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>.

- *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0.* Edited by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>.

Abstract:

This document defines data types, functions and other basic components of the PKCS #11 Cryptoki interface.

Status:

This document was last revised or approved by the OASIS PKCS 11 TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=pkcs11#technical.

TC members should send comments on this document to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at <https://www.oasis-open.org/committees/pkcs11/>.

This specification is provided under the [RF on RAND Terms](#) Mode of the [OASIS IPR Policy](#), the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

Note that any machine-readable content ([Computer Language Definitions](#)) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[PKCS11-Current-v3.0]

PKCS #11 Cryptographic Token Interface Current Mechanisms Specification Version 3.0. Edited by Chris Zimman and Dieter Bong. 29 May 2019. OASIS Committee Specification Draft 01 / Public Review Draft 01. <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/csprd01/pkcs11-curr-v3.0-csprd01.html>. Latest version: <https://docs.oasis-open.org/pkcs11/pkcs11-curr/v3.0/pkcs11-curr-v3.0.html>.

Notices

Copyright © OASIS Open 2019. All Rights Reserved.

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full [Policy](#) may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS **DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.**

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of [OASIS](#), the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark> for above guidance.

Table of Contents

1	Introduction	16
1.1	IPR Policy	16
1.2	Terminology	16
1.3	Definitions	16
1.4	Normative References	18
1.5	Non-Normative References	19
2	Mechanisms	22
2.1	RSA.....	22
2.1.1	Definitions.....	23
2.1.2	RSA public key objects.....	24
2.1.3	RSA private key objects	25
2.1.4	PKCS #1 RSA key pair generation	26
2.1.5	X9.31 RSA key pair generation.....	27
2.1.6	PKCS #1 v1.5 RSA	27
2.1.7	PKCS #1 RSA OAEP mechanism parameters	28
2.1.8	PKCS #1 RSA OAEP	30
2.1.9	PKCS #1 RSA PSS mechanism parameters	30
2.1.10	PKCS #1 RSA PSS	31
2.1.11	ISO/IEC 9796 RSA.....	31
2.1.12	X.509 (raw) RSA	32
2.1.13	ANSI X9.31 RSA	33
2.1.14	PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE-MD 128 or RIPE-MD 160	34
2.1.15	PKCS #1 v1.5 RSA signature with SHA-224	34
2.1.16	PKCS #1 RSA PSS signature with SHA-224	34
2.1.17	PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512.....	35
2.1.18	PKCS #1 v1.5 RSA signature with SHA3.....	35
2.1.19	PKCS #1 RSA PSS signature with SHA3	35
2.1.20	ANSI X9.31 RSA signature with SHA-1	35
2.1.21	TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA	36
2.1.22	TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP	36
2.1.23	RSA AES KEY WRAP	37
2.1.24	RSA AES KEY WRAP mechanism parameters	38
2.1.25	FIPS 186-4	39
2.2	DSA.....	39
2.2.1	Definitions	39
2.2.2	DSA public key objects.....	40
2.2.3	DSA Key Restrictions	41
2.2.4	DSA private key objects	41
2.2.5	DSA domain parameter objects	42
2.2.6	DSA key pair generation	43
2.2.7	DSA domain parameter generation.....	43
2.2.8	DSA probabilistic domain parameter generation.....	44
2.2.9	Shawe-Taylor domain parameter generation	44

2.2.10 DSA base domain parameter generation.....	44
2.2.11 DSA without hashing	45
2.2.12 DSA with SHA-1	45
2.2.13 FIPS 186-4	45
2.2.14 DSA with SHA-224	46
2.2.15 DSA with SHA-256	46
2.2.16 DSA with SHA-384	46
2.2.17 DSA with SHA-512	47
2.2.18 DSA with SHA3-224	47
2.2.19 DSA with SHA3-256	48
2.2.20 DSA with SHA3-384	48
2.2.21 DSA with SHA3-512	48
2.3 Elliptic Curve	49
2.3.1 EC Signatures	51
2.3.2 Definitions	51
2.3.3 ECDSA public key objects.....	52
2.3.4 Elliptic curve private key objects	53
2.3.5 Edwards Elliptic curve public key objects.....	55
2.3.6 Edwards Elliptic curve private key objects	55
2.3.7 Montgomery Elliptic curve public key objects.....	56
2.3.8 Montgomery Elliptic curve private key objects	57
2.3.9 Elliptic curve key pair generation.....	58
2.3.10 Edwards Elliptic curve key pair generation	59
2.3.11 Montgomery Elliptic curve key pair generation	59
2.3.12 ECDSA without hashing	60
2.3.13 ECDSA with hashing	60
2.3.14 EdDSA.....	61
2.3.15 For this mechanism, the <i>ulMinKeySize</i> and <i>ulMaxKeySize</i> fields of the CK_MECHANISM_INFO structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For this mechanism, the only allowed values are 255 and 448 as RFC 8032 and RFC 8410 only define curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the <i>ulMinKeySize</i> and <i>ulMaxKeySize</i> fields accordingly.XEdDSA	61
2.3.16 EC mechanism parameters.....	62
2.3.17 Elliptic curve Diffie-Hellman key derivation	67
2.3.18 Elliptic curve Diffie-Hellman with cofactor key derivation	68
2.3.19 Elliptic curve Menezes-Qu-Vanstone key derivation.....	68
2.3.20 ECDH AES KEY WRAP	69
2.3.21 ECDH AES KEY WRAP mechanism parameters	70
2.3.22 FIPS 186-4	71
2.4 Diffie-Hellman	71
2.4.1 Definitions	72
2.4.2 Diffie-Hellman public key objects	72
2.4.3 X9.42 Diffie-Hellman public key objects	73
2.4.4 Diffie-Hellman private key objects	74
2.4.5 X9.42 Diffie-Hellman private key objects	74
2.4.6 Diffie-Hellman domain parameter objects	75

2.4.7 X9.42 Diffie-Hellman domain parameters objects.....	76
2.4.8 PKCS #3 Diffie-Hellman key pair generation	77
2.4.9 PKCS #3 Diffie-Hellman domain parameter generation	77
2.4.10 PKCS #3 Diffie-Hellman key derivation.....	77
2.4.11 X9.42 Diffie-Hellman mechanism parameters.....	78
2.4.12 X9.42 Diffie-Hellman key pair generation.....	81
2.4.13 X9.42 Diffie-Hellman domain parameter generation	82
2.4.14 X9.42 Diffie-Hellman key derivation	82
2.4.15 X9.42 Diffie-Hellman hybrid key derivation	82
2.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation.....	83
2.5 Extended Triple Diffie-Hellman (x3dh).....	84
2.5.1 Definitions.....	84
2.5.2 Extended Triple Diffie-Hellman key objects	84
2.5.3 Initiating an Extended Triple Diffie-Hellman key exchange.....	84
2.5.4 Responding to an Extended Triple Diffie-Hellman key exchange.....	85
2.5.5 Extended Triple Diffie-Hellman parameters	86
2.6 Double Ratchet	86
2.6.1 Definitions.....	87
2.6.2 Double Ratchet secret key objects	87
2.6.3 Double Ratchet key derivation	88
2.6.4 Double Ratchet Encryption mechanism	90
2.6.5 Double Ratchet parameters	90
2.7 Wrapping/unwrapping private keys	90
2.8 Generic secret key	93
2.8.1 Definitions.....	93
2.8.2 Generic secret key objects	93
2.8.3 Generic secret key generation	94
2.9 HMAC mechanisms	94
2.9.1 General block cipher mechanism parameters.....	94
2.10 AES.....	94
2.10.1 Definitions.....	95
2.10.2 AES secret key objects	95
2.10.3 AES key generation.....	96
2.10.4 AES-ECB.....	96
2.10.5 AES-CBC.....	97
2.10.6 AES-CBC with PKCS padding	98
2.10.7 AES-OFB.....	98
2.10.8 AES-CFB	99
2.10.9 General-length AES-MAC	99
2.10.10 AES-MAC	99
2.10.11 AES-XCBC-MAC	100
2.10.12 AES-XCBC-MAC-96.....	100
2.11 AES with Counter	100
2.11.1 Definitions.....	100
2.11.2 AES with Counter mechanism parameters	101
2.11.3 AES with Counter Encryption / Decryption.....	101

2.12 AES CBC with Cipher Text Stealing CTS.....	101
2.12.1 Definitions.....	102
2.12.2 AES CTS mechanism parameters	102
2.13 Additional AES Mechanisms	102
2.13.1 Definitions.....	102
2.13.2 AES-GCM Authenticated Encryption / Decryption	103
2.13.3 AES-CCM authenticated Encryption / Decryption.....	104
2.13.4 AES-GMAC	106
2.13.5 AES GCM and CCM Mechanism parameters.....	107
2.14 AES CMAC	110
2.14.1 Definitions.....	110
2.14.2 Mechanism parameters.....	110
2.14.3 General-length AES-CMAC.....	110
2.14.4 AES-CMAC.....	110
2.15 AES XTS.....	111
2.15.1 Definitions.....	111
2.15.2 AES-XTS secret key objects	111
2.15.3 AES-XTS key generation	111
2.15.4 AES-XTS	112
2.16 AES Key Wrap.....	112
2.16.1 Definitions.....	112
2.16.2 AES Key Wrap Mechanism parameters.....	112
2.16.3 AES Key Wrap	112
2.17 Key derivation by data encryption – DES & AES	113
2.17.1 Definitions.....	113
2.17.2 Mechanism Parameters	114
2.17.3 Mechanism Description	114
2.18 Double and Triple-length DES	114
2.18.1 Definitions.....	115
2.18.2 DES2 secret key objects	115
2.18.3 DES3 secret key objects	116
2.18.4 Double-length DES key generation.....	116
2.18.5 Triple-length DES Order of Operations	117
2.18.6 Triple-length DES in CBC Mode	117
2.18.7 DES and Triple length DES in OFB Mode	117
2.18.8 DES and Triple length DES in CFB Mode.....	118
2.19 Double and Triple-length DES CMAC	118
2.19.1 Definitions.....	118
2.19.2 Mechanism parameters	119
2.19.3 General-length DES3-MAC	119
2.19.4 DES3-CMAC	119
2.20 SHA-1	119
2.20.1 Definitions.....	120
2.20.2 SHA-1 digest	120
2.20.3 General-length SHA-1-HMAC	120
2.20.4 SHA-1-HMAC	121

2.20.5 SHA-1 key derivation.....	121
2.20.6 SHA-1 HMAC key generation.....	121
2.21 SHA-224	122
2.21.1 Definitions.....	122
2.21.2 SHA-224 digest	122
2.21.3 General-length SHA-224-HMAC	123
2.21.4 SHA-224-HMAC	123
2.21.5 SHA-224 key derivation.....	123
2.21.6 SHA-224 HMAC key generation.....	123
2.22 SHA-256	124
2.22.1 Definitions.....	124
2.22.2 SHA-256 digest	124
2.22.3 General-length SHA-256-HMAC	124
2.22.4 SHA-256-HMAC	125
2.22.5 SHA-256 key derivation.....	125
2.22.6 SHA-256 HMAC key generation.....	125
2.23 SHA-384	125
2.23.1 Definitions.....	126
2.23.2 SHA-384 digest	126
2.23.3 General-length SHA-384-HMAC	126
2.23.4 SHA-384-HMAC	126
2.23.5 SHA-384 key derivation.....	127
2.23.6 SHA-384 HMAC key generation.....	127
2.24 SHA-512	127
2.24.1 Definitions.....	127
2.24.2 SHA-512 digest	127
2.24.3 General-length SHA-512-HMAC	128
2.24.4 SHA-512-HMAC	128
2.24.5 SHA-512 key derivation.....	128
2.24.6 SHA-512 HMAC key generation.....	128
2.25 SHA-512/224	129
2.25.1 Definitions.....	129
2.25.2 SHA-512/224 digest	129
2.25.3 General-length SHA-512/224-HMAC	129
2.25.4 SHA-512/224-HMAC	130
2.25.5 SHA-512/224 key derivation.....	130
2.25.6 SHA-512/224 HMAC key generation	130
2.26 SHA-512/256	130
2.26.1 Definitions.....	131
2.26.2 SHA-512/256 digest	131
2.26.3 General-length SHA-512/256-HMAC	131
2.26.4 SHA-512/256-HMAC	132
2.26.5 SHA-512/256 key derivation.....	132
2.26.6 SHA-512/256 HMAC key generation	132
2.27 SHA-512/t	132
2.27.1 Definitions.....	133

2.27.2 SHA-512/t digest	133
2.27.3 General-length SHA-512/t-HMAC	133
2.27.4 SHA-512/t-HMAC	133
2.27.5 SHA-512/t key derivation.....	134
2.27.6 SHA-512/t HMAC key generation	134
2.28 SHA3-224	134
2.28.1 Definitions	134
2.28.2 SHA3-224 digest	135
2.28.3 General-length SHA3-224-HMAC	135
2.28.4 SHA3-224-HMAC	135
2.28.5 SHA3-224 key derivation.....	135
2.28.6 SHA3-224 HMAC key generation	135
2.29 SHA3-256	136
2.29.1 Definitions	136
2.29.2 SHA3-256 digest	136
2.29.3 General-length SHA3-256-HMAC	136
2.29.4 SHA3-256-HMAC	137
2.29.5 SHA3-256 key derivation.....	137
2.29.6 SHA3-256 HMAC key generation	137
2.30 SHA3-384	137
2.30.1 Definitions	138
2.30.2 SHA3-384 digest	138
2.30.3 General-length SHA3-384-HMAC	138
2.30.4 SHA3-384-HMAC	139
2.30.5 SHA3-384 key derivation.....	139
2.30.6 SHA3-384 HMAC key generation	139
2.31 SHA3-512	139
2.31.1 Definitions	140
2.31.2 SHA3-512 digest	140
2.31.3 General-length SHA3-512-HMAC	140
2.31.4 SHA3-512-HMAC	140
2.31.5 SHA3-512 key derivation.....	141
2.31.6 SHA3-512 HMAC key generation	141
2.32 SHAKE.....	141
2.32.1 Definitions	141
2.32.2 SHAKE Key Derivation.....	141
2.33 Blake2b-160.....	142
2.33.1 Definitions	142
2.33.2 BLAKE2B-160 digest.....	142
2.33.3 General-length BLAKE2B-160-HMAC	143
2.33.4 BLAKE2B-160-HMAC	143
2.33.5 BLAKE2B-160 key derivation.....	143
2.33.6 BLAKE2B-160 HMAC key generation.....	143
2.34 BLAKE2B-256.....	143
2.34.1 Definitions	144
2.34.2 BLAKE2B-256 digest.....	144

2.34.3 General-length BLAKE2B-256-HMAC	144
2.34.4 BLAKE2B-256-HMAC	145
2.34.5 BLAKE2B-256 key derivation.....	145
2.34.6 BLAKE2B-256 HMAC key generation.....	145
2.35 BLAKE2B-384.....	145
2.35.1 Definitions.....	146
2.35.2 BLAKE2B-384 digest.....	146
2.35.3 General-length BLAKE2B-384-HMAC	146
2.35.4 BLAKE2B-384-HMAC	146
2.35.5 BLAKE2B-384 key derivation	147
2.35.6 BLAKE2B-384 HMAC key generation.....	147
2.36 BLAKE2B-512.....	147
2.36.1 Definitions.....	147
2.36.2 BLAKE2B-512 digest.....	147
2.36.3 General-length BLAKE2B-512-HMAC	148
2.36.4 BLAKE2B-512-HMAC	148
2.36.5 BLAKE2B-512 key derivation	148
2.36.6 BLAKE2B-512 HMAC key generation.....	148
2.37 PKCS #5 and PKCS #5-style password-based encryption (PBE).....	149
2.37.1 Definitions.....	149
2.37.2 Password-based encryption/authentication mechanism parameters.....	149
2.37.3 PKCS #5 PBKDF2 key generation mechanism parameters	150
2.37.4 PKCS #5 PBKD2 key generation	152
2.38 PKCS #12 password-based encryption/authentication mechanisms.....	152
2.38.1 SHA-1-PBE for 3-key triple-DES-CBC	153
2.38.2 SHA-1-PBE for 2-key triple-DES-CBC	153
2.38.3 SHA-1-PBA for SHA-1-HMAC.....	153
2.39 SSL	154
2.39.1 Definitions.....	154
2.39.2 SSL mechanism parameters.....	154
2.39.3 Pre-master key generation	156
2.39.4 Master key derivation	157
2.39.5 Master key derivation for Diffie-Hellman	157
2.39.6 Key and MAC derivation.....	158
2.39.7 MD5 MACing in SSL 3.0	159
2.39.8 SHA-1 MACing in SSL 3.0	159
2.40 TLS 1.2 Mechanisms	159
2.40.1 Definitions.....	160
2.40.2 TLS 1.2 mechanism parameters	160
◆ CK_TLS12_MASTER_KEY_DERIVE_PARAMS; CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR.....	160
◆ CK_TLS12_KEY_MAT_PARAMS; CK_TLS12_KEY_MAT_PARAMS_PTR.....	161
◆ CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR.....	161
◆ CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR	162
2.40.3 TLS MAC	163
2.40.4 Master key derivation	163

2.40.5 Master key derivation for Diffie-Hellman	164
2.40.6 Key and MAC derivation.....	165
2.40.7 CKM_TLS12_KEY_SAFE_DERIVE.....	166
2.40.8 Generic Key Derivation using the TLS PRF.....	166
2.40.9 Generic Key Derivation using the TLS12 PRF.....	167
2.41 WTLS	167
2.41.1 Definitions.....	168
2.41.2 WTLS mechanism parameters.....	168
2.41.3 Pre master secret key generation for RSA key exchange suite.....	171
2.41.4 Master secret key derivation	171
2.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography	172
2.41.6 WTLS PRF (pseudorandom function).....	173
2.41.7 Server Key and MAC derivation	173
2.41.8 Client key and MAC derivation	174
2.42 SP 800-108 Key Derivation	175
2.42.1 Definitions.....	175
2.42.2 Mechanism Parameters	176
2.42.3 Counter Mode KDF	181
2.42.4 Feedback Mode KDF	182
2.42.5 Double Pipeline Mode KDF	182
2.42.6 Deriving Additional Keys	183
2.42.7 Key Derivation Attribute Rules	184
2.42.8 Constructing PRF Input Data	184
2.42.8.1 Sample Counter Mode KDF.....	185
2.42.8.2 Sample SCP03 Counter Mode KDF.....	186
2.42.8.3 Sample Feedback Mode KDF	187
2.42.8.4 Sample Double-Pipeline Mode KDF	188
2.43 Miscellaneous simple key derivation mechanisms.....	189
2.43.1 Definitions.....	189
2.43.2 Parameters for miscellaneous simple key derivation mechanisms.....	189
2.43.3 Concatenation of a base key and another key.....	190
2.43.4 Concatenation of a base key and data.....	191
2.43.5 Concatenation of data and a base key.....	191
2.43.6 XORing of a key and data	192
2.43.7 Extraction of one key from another key.....	193
2.44 CMS.....	194
2.44.1 Definitions.....	194
2.44.2 CMS Signature Mechanism Objects	194
2.44.3 CMS mechanism parameters.....	194
2.44.4 CMS signatures	196
2.45 Blowfish.....	196
2.45.1 Definitions.....	197
2.45.2 BLOWFISH secret key objects.....	197
2.45.3 Blowfish key generation	198
2.45.4 Blowfish-CBC	198
2.45.5 Blowfish-CBC with PKCS padding	198

2.46 Twofish.....	199
2.46.1 Definitions.....	199
2.46.2 Twofish secret key objects	199
2.46.3 Twofish key generation	200
2.46.4 Twofish -CBC	200
2.46.5 Twofish-CBC with PKCS padding	200
2.47 CAMELLIA	200
2.47.1 Definitions.....	201
2.47.2 Camellia secret key objects.....	201
2.47.3 Camellia key generation	202
2.47.4 Camellia-ECB	202
2.47.5 Camellia-CBC.....	203
2.47.6 Camellia-CBC with PKCS padding	204
2.47.7 CAMELLIA with Counter mechanism parameters.....	204
2.47.8 General-length Camellia-MAC	205
2.47.9 Camellia-MAC	205
2.48 Key derivation by data encryption - Camellia	206
2.48.1 Definitions.....	206
2.48.2 Mechanism Parameters	206
2.49 ARIA.....	206
2.49.1 Definitions.....	207
2.49.2 Aria secret key objects	207
2.49.3 ARIA key generation	208
2.49.4 ARIA-ECB.....	208
2.49.5 ARIA-CBC	208
2.49.6 ARIA-CBC with PKCS padding	209
2.49.7 General-length ARIA-MAC	210
2.49.8 ARIA-MAC	210
2.50 Key derivation by data encryption - ARIA.....	210
2.50.1 Definitions.....	211
2.50.2 Mechanism Parameters	211
2.51 SEED	211
2.51.1 Definitions.....	212
2.51.2 SEED secret key objects.....	212
2.51.3 SEED key generation	213
2.51.4 SEED-ECB	213
2.51.5 SEED-CBC	213
2.51.6 SEED-CBC with PKCS padding.....	213
2.51.7 General-length SEED-MAC.....	214
2.51.8 SEED-MAC.....	214
2.52 Key derivation by data encryption - SEED	214
2.52.1 Definitions.....	214
2.52.2 Mechanism Parameters	214
2.53 OTP.....	214
2.53.1 Usage overview	214
2.53.2 Case 1: Generation of OTP values	215

2.53.3 Case 2: Verification of provided OTP values	216
2.53.4 Case 3: Generation of OTP keys	216
2.53.5 OTP objects.....	217
2.53.5.1 Key objects	217
2.53.6 OTP-related notifications	220
2.53.7 OTP mechanisms.....	220
2.53.7.1 OTP mechanism parameters	220
2.53.8 RSA SecurID	224
2.53.8.1 RSA SecurID secret key objects.....	224
2.53.8.2 RSA SecurID key generation	225
2.53.8.3 SecurID OTP generation and validation.....	226
2.53.8.4 Return values.....	226
2.53.9 OATH HOTP.....	226
2.53.9.1 OATH HOTP secret key objects	226
2.53.9.2 HOTP key generation	227
2.53.9.3 HOTP OTP generation and validation.....	227
2.53.10 ActivIdentity ACTI.....	227
2.53.10.1 ACTI secret key objects	227
2.53.10.2 ACTI key generation	228
2.53.10.3 ACTI OTP generation and validation	228
2.54 CT-KIP	229
2.54.1 Principles of Operation	229
2.54.2 Mechanisms	229
2.54.3 Definitions.....	230
2.54.4 CT-KIP Mechanism parameters	230
2.54.5 CT-KIP key derivation	230
2.54.6 CT-KIP key wrap and key unwrap.....	231
2.54.7 CT-KIP signature generation.....	231
2.55 GOST 28147-89	231
2.55.1 Definitions.....	232
2.55.2 GOST 28147-89 secret key objects	232
2.55.3 GOST 28147-89 domain parameter objects	233
2.55.4 GOST 28147-89 key generation	233
2.55.5 GOST 28147-89-ECB	234
2.55.6 GOST 28147-89 encryption mode except ECB	234
2.55.7 GOST 28147-89-MAC	235
2.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89	235
2.56 GOST R 34.11-94	236
2.56.1 Definitions.....	236
2.56.2 GOST R 34.11-94 domain parameter objects.....	236
2.56.3 GOST R 34.11-94 digest.....	237
2.56.4 GOST R 34.11-94 HMAC	238
2.57 GOST R 34.10-2001	238
2.57.1 Definitions.....	239
2.57.2 GOST R 34.10-2001 public key objects	239
2.57.3 GOST R 34.10-2001 private key objects	240
2.57.4 GOST R 34.10-2001 domain parameter objects.....	242

2.57.5 GOST R 34.10-2001 mechanism parameters.....	243
2.57.6 GOST R 34.10-2001 key pair generation.....	244
2.57.7 GOST R 34.10-2001 without hashing	245
2.57.8 GOST R 34.10-2001 with GOST R 34.11-94.....	245
2.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001	246
2.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys	246
2.58 ChaCha20.....	246
2.58.1 Definitions	246
2.58.2 ChaCha20 secret key objects	247
2.58.3 ChaCha20 mechanism parameters	247
2.58.4 ChaCha20 key generation.....	248
2.58.5 ChaCha20 mechanism	248
2.59 Salsa20	249
2.59.1 Definitions	249
2.59.2 Salsa20 secret key objects.....	249
2.59.3 Salsa20 mechanism parameters.....	250
2.59.4 Salsa20 key generation	250
2.59.5 Salsa20 mechanism	251
2.60 Poly1305	251
2.60.1 Definitions	252
2.60.2 Poly1305 secret key objects.....	252
2.60.3 Poly1305 mechanism	252
2.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption	253
2.61.1 Definitions	253
2.61.2 Usage	253
2.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters	254
2.62 HKDF Mechanisms.....	255
2.62.1 Definitions	256
2.62.2 HKDF mechanism parameters	256
2.62.3 HKDF derive	257
2.62.4 HKDF Data	258
2.62.5 HKDF Key gen	258
2.63 NULL Mechanism	258
2.63.1 Definitions	258
2.63.2 CKM_NULL mechanism parameters	258
3 PKCS #11 Implementation Conformance	259
Appendix A. Acknowledgments	260
Appendix B. Manifest Constants	263
B.1 Object classes	263
B.2 Key types.....	263
B.3 Key derivation functions	264
B.4 Mechanisms	265
B.5 Attributes	274
B.6 Attribute constants.....	276
B.7 Other constants	276
B.8 Notifications	277

B.9 Return values	277
Appendix C. Revision History	280

1 Introduction

2 This document defines mechanisms that are anticipated to be used with the current version of PKCS #11.
3 All text is normative unless otherwise labeled.

4 1.1 IPR Policy

5 This specification is provided under the [RF on RAND Terms](#) Mode of the [OASIS IPR Policy](#), the mode
6 chosen when the Technical Committee was established. For information on whether any patents have
7 been disclosed that may be essential to implementing this specification, and any offers of patent licensing
8 terms, please refer to the Intellectual Property Rights section of the TC's web page (<https://www.oasis-open.org/committees/pkcs11/ipr.php>).

10 1.2 Terminology

11 The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD
12 NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described
13 in [RFC2119]

14 1.3 Definitions

15 For the purposes of this standard, the following definitions apply. Please refer to the [PKCS#11-Base] for
16 further definitions:

17	AES	<i>Advanced Encryption Standard, as defined in FIPS PUB 197.</i>
18	CAMELLIA	<i>The Camellia encryption algorithm, as defined in RFC 3713.</i>
19	BLOWFISH	<i>The Blowfish Encryption Algorithm of Bruce Schneier, www.schneier.com.</i>
21	CBC	<i>Cipher-Block Chaining mode, as defined in FIPS PUB 81.</i>
22	CDMF	<i>Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.</i>
25	CMAC	<i>Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].</i>
27	CMS	<i>Cryptographic Message Syntax (see RFC 2630)</i>
28	CT-KIP	<i>Cryptographic Token Key Initialization Protocol (as defined in [CT- KIP])</i>
30	DES	<i>Data Encryption Standard, as defined in FIPS PUB 46-3.</i>
31	DSA	<i>Digital Signature Algorithm, as defined in FIPS PUB 186-2.</i>
32	EC	<i>Elliptic Curve</i>
33	ECB	<i>Electronic Codebook mode, as defined in FIPS PUB 81.</i>
34	ECDH	<i>Elliptic Curve Diffie-Hellman.</i>

35	ECDSA	<i>Elliptic Curve DSA, as in ANSI X9.62.</i>
36	ECMQV	<i>Elliptic Curve Menezes-Qu-Vanstone</i>
37	GOST 28147-89	The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].
38		
39	GOST R 34.11-94	<i>Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].</i>
40		
41	GOST R 34.10-2001	<i>The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491].</i>
42		
43	IV	<i>Initialization Vector.</i>
44	MAC	<i>Message Authentication Code.</i>
45	MQV	<i>Menezes-Qu-Vanstone</i>
46	OAEP	<i>Optimal Asymmetric Encryption Padding for RSA.</i>
47	PKCS	<i>Public-Key Cryptography Standards.</i>
48	PRF	<i>Pseudo random function.</i>
49	PTD	<i>Personal Trusted Device, as defined in MeT-PTD</i>
50	RSA	<i>The RSA public-key cryptosystem.</i>
51	SHA-1	<i>The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.</i>
52		
53	SHA-224	<i>The Secure Hash Algorithm with a 224-bit message digest, as defined in RFC 3874. Also defined in FIPS PUB 180-2 with Change Notice 1.</i>
54		
55		
56	SHA-256	<i>The Secure Hash Algorithm with a 256-bit message digest, as defined in FIPS PUB 180-2.</i>
57		
58	SHA-384	<i>The Secure Hash Algorithm with a 384-bit message digest, as defined in FIPS PUB 180-2.</i>
59		
60	SHA-512	<i>The Secure Hash Algorithm with a 512-bit message digest, as defined in FIPS PUB 180-2.</i>
61		
62	SSL	<i>The Secure Sockets Layer 3.0 protocol.</i>
63	SO	<i>A Security Officer user.</i>
64	TLS	<i>Transport Layer Security.</i>
65	WIM	<i>Wireless Identification Module.</i>
66	WTLS	<i>Wireless Transport Layer Security.</i>
67		

68 **1.4 Normative References**

- 69 [ARIA] National Security Research Institute, Korea, "Block Cipher Algorithm ARIA",
70 URL: <http://tools.ietf.org/html/rfc5794>
- 71 [BLOWFISH] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher
72 (Blowfish), December 1993.
73 URL: <https://www.schneier.com/paper-blowfish-fse.html>
- 74 [CAMELLIA] M. Matsui, J. Nakajima, S. Moriai. A Description of the Camellia Encryption
75 Algorithm, April 2004.
76 URL: <http://www.ietf.org/rfc/rfc3713.txt>
- 77 [CDMF] Johnson, D.B. The Commercial Data Masking Facility (CDMF) data privacy
78 algorithm, March 1994.
79 URL: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5389557>
- 80 [CHACHA] D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.
81 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>
- 82 [DH] W. Diffie, M. Hellman. New Directions in Cryptography. Nov, 1976.
83 URL: <http://www-ee.stanford.edu/~hellman/publications/24.pdf>
- 84 [FIPS PUB 81] NIST. FIPS 81: DES Modes of Operation. December 1980.
85 URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>
- 86 [FIPS PUB 186-4] NIST. FIPS 186-4: Digital Signature Standard. July 2013.
87 URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>
- 88 [FIPS PUB 197] NIST. FIPS 197: Advanced Encryption Standard. November 26, 2001.
89 URL: <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>
- 90 [FIPS SP 800-56A] NIST. Special Publication 800-56A Revision 2: *Recommendation for Pair-Wise
91 Key Establishment Schemes Using Discrete Logarithm Cryptography*, May 2013.
92 URL: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>
- 93 [FIPS SP 800-108] NIST. Special Publication 800-108 (Revised): *Recommendation for Key
94 Derivation Using Pseudorandom Functions*, October 2009.
95 URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>
- 96 [GOST] V. Dolmatov, A. Degtyarev. GOST R. 34.11-2012: Hash Function. August 2013.
97 URL: <http://tools.ietf.org/html/rfc6986>
- 98 [MD2] B. Kaliski. RSA Laboratories. The MD2 Message-Digest Algorithm. April, 1992.
99 URL: <http://tools.ietf.org/html/rfc1319>
- 100 [MD5] RSA Data Security. R. Rivest. The MD5 Message-Digest Algorithm. April, 1992.
101 URL: <http://tools.ietf.org/html/rfc1319>
- 102 [OAEP] M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to Encrypt with
103 RSA. Nov 19, 1995.
104 URL: <http://cseweb.ucsd.edu/users/mihir/papers/oaep.pdf>
- 105 [PKCS11-Base] PKCS #11 Cryptographic Token Interface Base Specification Version 3.0. Edited
106 by Chris Zimman and Dieter Bong. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-base/v3.0/pkcs11-base-v3.0.html>.
- 107 [PKCS11-Hist] PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification
108 Version 3.0. Edited by Chris Zimman and Dieter Bong. Latest version.
<https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>.
- 109 [PKCS11-Prof] PKCS #11 Cryptographic Token Interface Profiles Version 3.0. Edited by Tim
110 Hudson. Latest version. <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.0/pkcs11-profiles-v3.0.html>.
- 111 [POLY1305] D.J. Bernstein. The Poly1305-AES message-authentication code. Jan 2005.
112 URL: <https://cr.yp.to/mac/poly1305-20050329.pdf>
- 113 [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP
114 14, RFC 2119, March 1997.
115 URL: <http://www.ietf.org/rfc/rfc2119.txt>.

120	[RIPEMD]	H. Dobbertin, A. Bosselaers, B. Preneel. The hash function RIPEMD-160, Feb 13, 2012. URL: http://homes.esat.kuleuven.be/~bosselaer/ripemd160.html
121		
122		
123	[SALSA]	D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008. URL: http://cr.yp.to/chacha/chacha-20080128.pdf
124		
125	[SEED]	KISA. SEED 128 Algorithm Specification. Sep 2003. URL: http://seed.kisa.or.kr/html/egovframework/iwt/ds/ko/ref/%5B2%5D_SEED+128_Specification_english_M.pdf
126		
127		
128	[SHA-1]	NIST. FIPS 180-4: Secure Hash Standard. March 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
129		
130	[SHA-2]	NIST. FIPS 180-4: Secure Hash Standard. March 2012. URL: http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf
131		
132	[TWOFISH]	B. Schneier, J. Kelsey, D. Whiting, C. Hall, N. Ferguson. Twofish: A 128-Bit Block Cipher. June 15, 1998. URL: https://www.schneier.com/paper-twofish-paper.pdf
133		
134		

1.5 Non-Normative References

135	[CAP-1.2]	<i>Common Alerting Protocol Version 1.2.</i> 01 July 2010. OASIS Standard. URL: http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html
136		
137		
138	[AES KEYWRAP]	National Institute of Standards and Technology, NIST Special Publication 800-38F, Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping, December 2012, http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf
139		
140		
141		
142	[ANSI C]	ANSI/ISO. American National Standard for Programming Languages – C. 1990.
143	[ANSI X9.31]	Accredited Standards Committee X9. Digital Signatures Using Reversible Public Key Cryptography for the Financial Services Industry (rDSA). 1998.
144		
145	[ANSI X9.42]	Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. 2003.
146		
147		
148	[ANSI X9.62]	Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). 1998.
149		
150	[ANSI X9.63]	Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. 2001. URL: http://webstore.ansi.org/RecordDetail.aspx?sku=X9.63-2011
151		
152		
153		
154	[BRAINPOOL]	ECC Brainpool Standard Curves and Curve Generation, v1.0, 19.10.2005 URL: http://www.ecc-brainpool.org
155		
156	[CT-KIP]	RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0, December 2005. URL: ftp://ftp.rsasecurity.com/pub/otps/ct-kip/ct-kip-v1-0.pdf .
157		
158		
159	[CC/PP]	CCPP-STRUCT-VOCAB, G. Klyne, F. Reynolds, C. , H. Ohto, J. Hjelm, M. H. Butler, L. Tran, Editors, W3C Recommendation, 15 January 2004, URL: http://www.w3.org/TR/2004/REC-CCPP-struct-vocab-20040115/ Latest version available at http://www.w3.org/TR/CCPP-struct-vocab/
160		
161		
162		
163	[LEGIFRANCE]	Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français (Publication of elliptic curve parameters by the French state) URL: https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816
164		
165		
166		
167		
168	[NIST AES CTS]	National Institute of Standards and Technology, Addendum to NIST Special Publication 800-38A, “Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC Mode”
169		
170		

- 171 URL: http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf
- 172
- 173 [PKCS11-UG] PKCS #11 Cryptographic Token Interface Usage Guide Version 2.41. Edited by John Leiseboer and Robert Griffin. version: <http://docs.oasis-open.org/pkcs11/pkcs11-ug/v2.40/pkcs11-ug-v2.40.html>.
- 174
- 175
- 176 [RFC 2865] Rigney et al, "Remote Authentication Dial In User Service (RADIUS)", IETF RFC2865, June 2000.
URL: <http://www.ietf.org/rfc/rfc2865.txt>.
- 177
- 178
- 179 [RFC 3686] Housley, "Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security Payload (ESP)," IETF RFC 3686, January 2004.
URL: <http://www.ietf.org/rfc/rfc3686.txt>.
- 180
- 181
- 182 [RFC 3717] Matsui, et al, "A Description of the Camellia Encryption Algorithm," IETF RFC 3717, April 2004.
URL: <http://www.ietf.org/rfc/rfc3713.txt>.
- 183
- 184
- 185 [RFC 3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", IETF RFC 3610, September 2003.
URL: <http://www.ietf.org/rfc/rfc3610.txt>
- 186
- 187
- 188 [RFC 3874] Smit et al, "A 224-bit One-way Hash Function: SHA-224," IETF RFC 3874, June 2004.
URL: <http://www.ietf.org/rfc/rfc3874.txt>.
- 189
- 190
- 191 [RFC 3748] Aboba et al, "Extensible Authentication Protocol (EAP)", IETF RFC 3748, June 2004.
URL: <http://www.ietf.org/rfc/rfc3748.txt>.
- 192
- 193
- 194 [RFC 4269] South Korean Information Security Agency (KISA) "The SEED Encryption Algorithm", December 2005.
URL: <ftp://ftp.rfc-editor.org/in-notes/rfc4269.txt>
- 195
- 196
- 197 [RFC 4309] Housley, R., "Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)," IETF RFC 4309, December 2005.
URL: <http://www.ietf.org/rfc/rfc4309.txt>
- 198
- 199
- 200 [RFC 4357] V. Popov, I. Kurepkin, S. Leontiev "Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms", January 2006.
URL: <http://www.ietf.org/rfc/rfc4357.txt>
- 201
- 202
- 203
- 204 [RFC 4490] S. Leontiev, Ed. G. Chudov, Ed. "Using the GOST 28147-89, GOST R 34.11-94,GOST R 34.10-94, and GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)", May 2006.
URL: <http://www.ietf.org/rfc/rfc4490.txt>
- 205
- 206
- 207
- 208 [RFC 4491] S. Leontiev, Ed., D. Shefanovski, Ed., "Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile", May 2006.
URL: <http://www.ietf.org/rfc/rfc4491.txt>
- 209
- 210
- 211
- 212 [RFC 4493] J. Song et al. *RFC 4493: The AES-CMAC Algorithm.* June 2006.
URL: <http://www.ietf.org/rfc/rfc4493.txt>
- 213
- 214 [RFC 5705] Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC 5705, March 2010.
URL: <http://www.ietf.org/rfc/rfc5705.txt>
- 215
- 216
- 217 [RFC 5869] H. Krawczyk, P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", May 2010
URL: <http://www.ietf.org/rfc/rfc5869.txt>
- 218
- 219
- 220 [RFC 7539] Y Nir, A. Langley. *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols,* May 2015
URL: <https://tools.ietf.org/rfc/rfc7539.txt>
- 221
- 222

223	[RFC 7748]	Aboba et al, "Elliptic Curves for Security", IETF RFC 7748, January 2016 URL: https://tools.ietf.org/html/rfc7748
224		
225	[RFC 8032]	Aboba et al, "Edwards-Curve Digital Signature Algorithm (EdDSA)", IETF RFC 8032, January 2017 URL: https://tools.ietf.org/html/rfc8032
226		
227		
228	[SEC 1]	Standards for Efficient Cryptography Group (SECG). <i>Standards for Efficient Cryptography (SEC) 1: Elliptic Curve Cryptography</i> . Version 1.0, September 20, 2000.
229		
230		
231	[SEC 2]	Standards for Efficient Cryptography Group (SECG). <i>Standards for Efficient Cryptography (SEC) 2: Recommended Elliptic Curve Domain Parameters</i> . Version 1.0, September 20, 2000.
232		
233		
234	[SIGNAL]	The X3DH Key Agreement Protocol, Revision 1, 2016-11-04, Moxie Marlinspike, Trevor Perrin (editor) URL: https://signal.org/docs/specifications/x3dh/
235		
236		
237	[TLS]	[RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999. http://www.ietf.org/rfc/rfc2246.txt , superseded by [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006. http://www.ietf.org/rfc/rfc4346.txt , which was superseded by [5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008. URL: http://www.ietf.org/rfc/rfc5246.txt
238		
239		
240		
241		
242		
243		
244	[TLS12]	[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008. URL: http://www.ietf.org/rfc/rfc5246.txt
245		
246		
247	[WIM]	WAP. Wireless Identity Module. — WAP-260-WIM-20010712-a. July 2001. URL: http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.aspx?DocName=/wap/wap-260-wim-20010712-a.pdf
248		
249		
250	[WPKI]	Wireless Application Protocol: Public Key Infrastructure Definition. — WAP-217-WPKI-20010424-a. April 2001. URL: http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.aspx?DocName=/wap/wap-217-wpki-20010424-a.pdf
251		
252		
253		
254	[WTLS]	WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001. URL: http://technical.openmobilealliance.org/tech/affiliates/LicenseAgreement.aspx?DocName=/wap/wap-261-wtls-20010406-a.pdf
255		
256		
257		
258	[XEDDSA]	The XEdDSA and VXEddSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin (editor) URL: https://signal.org/docs/specifications/xeddsa/
259		
260		
261	[X.500]	ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1
262		
263		
264	[X.509]	ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8
265		
266		
267	[X.680]	ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation. July 2002. Identical to ISO/IEC 8824-1
268		
269	[X.690]	ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical to ISO/IEC 8825-1
270		
271		
272		

273

2 Mechanisms

274 A mechanism specifies precisely how a certain cryptographic process is to be performed. PKCS #11
275 implementations MAY use one or more mechanisms defined in this document.

276 The following table shows which Cryptoki mechanisms are supported by different cryptographic
277 operations. For any particular token, of course, a particular operation may well support only a subset of
278 the mechanisms listed. There is also no guarantee that a token which supports one mechanism for some
279 operations supports any other mechanism for any other operation (or even supports that same
280 mechanism for any other operation). For example, even if a token is able to create RSA digital signatures
281 with the **CKM_RSA_PKCS** mechanism, it may or may not be the case that the same token can also
282 perform RSA encryption with **CKM_RSA_PKCS**.

283 Each mechanism description is preceded by a table, of the following format, mapping mechanisms to
284 API functions.

285

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive

286 1 SR = SignRecover, VR = VerifyRecover.

287 2 Single-part operations only.

288 3 Mechanism can only be used for wrapping, not unwrapping.

289 The remainder of this section will present in detail the mechanisms supported by Cryptoki and the parameters which are supplied to them.

290 In general, if a mechanism makes no mention of the ulMinKeyLen and ulMaxKeyLen fields of the CK_MECHANISM_INFO structure, then those fields have no
291 meaning for that particular mechanism.

292

2.1 RSA

293 *Table 1, Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_RSA_PKCS_KEY_PAIR_GEN					✓		
CKM_RSA_X9_31_KEY_PAIR_GEN					✓		
CKM_RSA_PKCS	✓ ²	✓ ²	✓			✓	
CKM_RSA_PKCS_OAEP	✓ ²					✓	
CKM_RSA_PKCS_PSS		✓ ²					
CKM_RSA_9796		✓ ²	✓				
CKM_RSA_X_509	✓ ²	✓ ²	✓			✓	
CKM_RSA_X9_31		✓ ²					
CKM_SHA1_RSA_PKCS		✓					
CKM_SHA256_RSA_PKCS		✓					
CKM_SHA384_RSA_PKCS		✓					

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ₁	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_RSA_PKCS		✓					
CKM_SHA1_RSA_PKCS_PSS		✓					
CKM_SHA256_RSA_PKCS_PSS		✓					
CKM_SHA384_RSA_PKCS_PSS		✓					
CKM_SHA512_RSA_PKCS_PSS		✓					
CKM_SHA1_RSA_X9_31		✓					
CKM_RSA_PKCS TPM_1_1	✓ ²					✓	
CKM_RSA_PKCS_OAEP TPM_1_1	✓ ²					✓	
CKM_SHA3_224_RSA_PKCS		✓					
CKM_SHA3_256_RSA_PKCS		✓					
CKM_SHA3_384_RSA_PKCS		✓					
CKM_SHA3_512_RSA_PKCS		✓					
CKM_SHA3_224_RSA_PKCS_PSS		✓					
CKM_SHA3_256_RSA_PKCS_PSS		✓					
CKM_SHA3_384_RSA_PKCS_PSS		✓					
CKM_SHA3_512_RSA_PKCS_PSS		✓					

294 2.1.1 Definitions

295 This section defines the RSA key type "CKK_RSA" for type CK_KEY_TYPE as used in the
 296 CKA_KEY_TYPE attribute of RSA key objects.

297 Mechanisms:

```

 298     CKM_RSA_PKCS_KEY_PAIR_GEN
 299     CKM_RSA_PKCS
 300     CKM_RSA_9796
 301     CKM_RSA_X_509
 302     CKM_MD2_RSA_PKCS
 303     CKM_MD5_RSA_PKCS
 304     CKM_SHA1_RSA_PKCS
 305     CKM_SHA224_RSA_PKCS
 306     CKM_SHA256_RSA_PKCS
 307     CKM_SHA384_RSA_PKCS
 308     CKM_SHA512_RSA_PKCS
 309     CKM_RIPEMD128_RSA_PKCS
 310     CKM_RIPEMD160_RSA_PKCS
 311     CKM_RSA_PKCS_OAEP
 312     CKM_RSA_X9_31_KEY_PAIR_GEN
 313     CKM_RSA_X9_31
 314     CKM_SHA1_RSA_X9_31
 315     CKM_RSA_PKCS_PSS
 316     CKM_SHA1_RSA_PKCS_PSS

```

```

317      CKM_SHA224_RSA_PKCS_PSS
318      CKM_SHA256_RSA_PKCS_PSS
319      CKM_SHA512_RSA_PKCS_PSS
320      CKM_SHA384_RSA_PKCS_PSS
321      CKM_RSA_PKCS TPM_1_1
322      CKM_RSA_PKCS_OAEP TPM_1_1
323      CKM_RSA_AES_KEY_WRAP
324      CKM_SHA3_224_RSA_PKCS
325      CKM_SHA3_256_RSA_PKCS
326      CKM_SHA3_384_RSA_PKCS
327      CKM_SHA3_512_RSA_PKCS
328      CKM_SHA3_224_RSA_PKCS_PSS
329      CKM_SHA3_256_RSA_PKCS_PSS
330      CKM_SHA3_384_RSA_PKCS_PSS
331      CKM_SHA3_512_RSA_PKCS_PSS
332

```

333 2.1.2 RSA public key objects

334 RSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_RSA**) hold RSA public keys.
 335 The following table defines the RSA public key object attributes, in addition to the common attributes
 336 defined for this object class:

337 *Table 2, RSA Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_MODULUS ^{1,4}	Big integer	Modulus n
CKA_MODULUS_BITS ^{2,3}	CK ULONG	Length in bits of modulus n
CKA_PUBLIC_EXPONENT ¹	Big integer	Public exponent e

338 - Refer to [PKCS11-Base] table 11 for footnotes

339 Depending on the token, there may be limits on the length of key components. See PKCS #1 for more
 340 information on RSA keys.

341 The following is a sample template for creating an RSA public key object:

```

342     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
343     CK_KEY_TYPE keyType = CKK_RSA;
344     CK_UTF8CHAR label[] = "An RSA public key object";
345     CK_BYTE modulus[] = {...};
346     CK_BYTE exponent[] = {...};
347     CK_BBOOL true = CK_TRUE;
348     CK_ATTRIBUTE template[] = {
349         {CKA_CLASS, &class, sizeof(class)},
350         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
351         {CKA_TOKEN, &true, sizeof(true)},
352         {CKA_LABEL, label, sizeof(label)-1},
353         {CKA_WRAP, &true, sizeof(true)},
354         {CKA_ENCRYPT, &true, sizeof(true)},
355         {CKA_MODULUS, modulus, sizeof(modulus)},

```

```
356     { CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent) }  
357 }
```

358 2.1.3 RSA private key objects

359 RSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_RSA**) hold RSA private keys.
360 The following table defines the RSA private key object attributes, in addition to the common attributes
361 defined for this object class:

362 *Table 3, RSA Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_MODULUS ^{1,4,6}	Big integer	Modulus n
CKA_PUBLIC_EXPONENT ^{4,6}	Big integer	Public exponent e
CKA_PRIVATE_EXPONENT ^{1,4,6,7}	Big integer	Private exponent d
CKA_PRIME_1 ^{4,6,7}	Big integer	Prime p
CKA_PRIME_2 ^{4,6,7}	Big integer	Prime q
CKA_EXPONENT_1 ^{4,6,7}	Big integer	Private exponent d modulo $p-1$
CKA_EXPONENT_2 ^{4,6,7}	Big integer	Private exponent d modulo $q-1$
CKA_COEFFICIENT ^{4,6,7}	Big integer	CRT coefficient $q^{-1} \bmod p$

363 - Refer to [PKCS11-Base] table 11 for footnotes

364 Depending on the token, there may be limits on the length of the key components. See PKCS #1 for
365 more information on RSA keys.

366 Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above
367 attributes, which can assist in performing rapid RSA computations. Other tokens might store only the
368 **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values. Effective with version 2.40, tokens MUST
369 also store **CKA_PUBLIC_EXPONENT**. This permits the retrieval of sufficient data to reconstitute the
370 associated public key.

371 Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an
372 RSA private key, it stores whichever of the fields in Table 3 it keeps track of. Later, if an application asks
373 for the values of the key's various attributes, Cryptoki supplies values only for attributes whose values it
374 can obtain (i.e., if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note
375 that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA
376 private keys which are not actually stored on the token. *E.g.*, if a particular token stores values only for
377 the **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then Cryptoki is
378 certainly *able* to report values for all the attributes above (since they can all be computed efficiently from
379 these three values). However, a Cryptoki implementation may or may not actually do this extra
380 computation. The only attributes from Table 3 for which a Cryptoki implementation is *required* to be able
381 to return values are **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT**.

382 If an RSA private key object is created on a token, and more attributes from Table 3 are supplied to the
383 object creation call than are supported by the token, the extra attributes are likely to be thrown away. If
384 an attempt is made to create an RSA private key object on a token with insufficient attributes for that
385 particular token, then the object creation call fails and returns CKR_TEMPLATE_INCOMPLETE.

386 Note that when generating an RSA private key, there is no **CKA_MODULUS_BITS** attribute specified.
387 This is because RSA private keys are only generated as part of an RSA key pair, and the
388 **CKA_MODULUS_BITS** attribute for the pair is specified in the template for the RSA public key.

389 The following is a sample template for creating an RSA private key object:

```
390 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
391 CK_KEY_TYPE keyType = CKK_RSA;  
392 CK_UTF8CHAR label[] = "An RSA private key object";  
393 CK_BYTE subject[] = {...};  
394 CK_BYTE id[] = {123};
```

```

395     CK_BYTE modulus[] = {...};
396     CK_BYTE publicExponent[] = {...};
397     CK_BYTE privateExponent[] = {...};
398     CK_BYTE prime1[] = {...};
399     CK_BYTE prime2[] = {...};
400     CK_BYTE exponent1[] = {...};
401     CK_BYTE exponent2[] = {...};
402     CK_BYTE coefficient[] = {...};
403     CK_BBOOL true = CK_TRUE;
404     CK_ATTRIBUTE template[] = {
405         {CKA_CLASS, &class, sizeof(class)},
406         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
407         {CKA_TOKEN, &true, sizeof(true)},
408         {CKA_LABEL, label, sizeof(label)-1},
409         {CKA SUBJECT, subject, sizeof(subject)},
410         {CKA_ID, id, sizeof(id)},
411         {CKA_SENSITIVE, &true, sizeof(true)},
412         {CKA_DECRYPT, &true, sizeof(true)},
413         {CKA_SIGN, &true, sizeof(true)},
414         {CKA_MODULUS, modulus, sizeof(modulus)},
415         {CKA_PUBLIC_EXPONENT, publicExponent,
416             sizeof(publicExponent)},
417         {CKA_PRIVATE_EXPONENT, privateExponent,
418             sizeof(privateExponent)},
419         {CKA_PRIME_1, prime1, sizeof(prime1)},
420         {CKA_PRIME_2, prime2, sizeof(prime2)},
421         {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
422         {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
423         {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}}
424     };

```

2.1.4 PKCS #1 RSA key pair generation

426 The PKCS #1 RSA key pair generation mechanism, denoted **CKM_RSA_PKCS_KEY_PAIR_GEN**, is a
427 key pair generation mechanism based on the RSA public-key cryptosystem, as defined in PKCS #1.

428 It does not have a parameter.

429 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
430 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
431 template for the public key. The **CKA_PUBLIC_EXPONENT** may be omitted in which case the
432 mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537).
433 Specific implementations may use a random value or an alternative default if 0x10001 cannot be used by
434 the token.

435 Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error
436 if this attribute is omitted from the template. Experience has shown that many implementations of 2.11
437 and prior did allow the **CKA_PUBLIC_EXPONENT** attribute to be omitted from the template, and
438 behaved as described above. The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**,
439 **CKA_MODULUS**, and **CKA_PUBLIC_EXPONENT** attributes to the new public key.
440 **CKA_PUBLIC_EXPONENT** will be copied from the template if supplied.

441 **CKR_TEMPLATE_INCONSISTENT** shall be returned if the implementation cannot use the supplied
442 exponent value. It contributes the **CKA_CLASS** and **CKA_KEY_TYPE** attributes to the new private key; it

443 may also contribute some of the following attributes to the new private key: **CKA_MODULUS**,
444 **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**,
445 **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**. Other attributes supported by the
446 RSA public and private key types (specifically, the flags indicating which functions the keys support) may
447 also be specified in the templates for the keys, or else are assigned default initial values.
448 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
449 specify the supported range of RSA modulus sizes, in bits.

450 2.1.5 X9.31 RSA key pair generation

451 The X9.31 RSA key pair generation mechanism, denoted **CKM_RSA_X9_31_KEY_PAIR_GEN**, is a key
452 pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

453 It does not have a parameter.

454 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
455 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
456 template for the public key.

457 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**, and
458 **CKA_PUBLIC_EXPONENT** attributes to the new public key. It contributes the **CKA_CLASS** and
459 **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes
460 to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**,
461 **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**.
462 Other attributes supported by the RSA public and private key types (specifically, the flags indicating which
463 functions the keys support) may also be specified in the templates for the keys, or else are assigned
464 default initial values. Unlike the **CKM_RSA_PKCS_KEY_PAIR_GEN** mechanism, this mechanism is
465 guaranteed to generate *p* and *q* values, **CKA_PRIME_1** and **CKA_PRIME_2** respectively, that meet the
466 strong primes requirement of X9.31.

467 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
468 specify the supported range of RSA modulus sizes, in bits.

469 2.1.6 PKCS #1 v1.5 RSA

470 The PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS**, is a multi-purpose mechanism based
471 on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports
472 single-part encryption and decryption; single-part signatures and verification with and without message
473 recovery; key wrapping; and key unwrapping. This mechanism corresponds only to the part of PKCS #1
474 v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for
475 the md2withRSAEncryption and md5withRSAEncryption algorithms in PKCS #1 v1.5 .

476 This mechanism does not have a parameter.

477 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
478 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
479 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
480 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
481 key, except the key length; the application must convey these separately. In particular, the mechanism
482 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
483 to the recovered key during unwrapping; other attributes must be specified in the template.

484 Constraints on key types and the length of the data are summarized in the following table. For
485 encryption, decryption, signatures and signature verification, the input and output data may begin at the
486 same location in memory. In the table, *k* is the length in bytes of the RSA modulus.

487 Table 4, PKCS #1 v1.5 RSA: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt ¹	RSA public key	$\leq k-11$	k	block type 02
C_Decrypt ¹	RSA private key	k	$\leq k-11$	block type 02
C_Sign ¹	RSA private key	$\leq k-11$	k	block type 01
C_SignRecover	RSA private key	$\leq k-11$	k	block type 01
C_Verify ¹	RSA public key	$\leq k-11, k^2$	N/A	block type 01
C_VerifyRecover	RSA public key	k	$\leq k-11$	block type 01
C_WrapKey	RSA public key	$\leq k-11$	k	block type 02
C_UnwrapKey	RSA private key	k	$\leq k-11$	block type 02

488 1 Single-part operations only.

489 2 Data length, signature length.

490 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
491 specify the supported range of RSA modulus sizes, in bits.492 **2.1.7 PKCS #1 RSA OAEP mechanism parameters**493 ◆ **CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR**494 **CK_RSA_PKCS_MGF_TYPE** is used to indicate the Message Generation Function (MGF) applied to a
495 message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the
496 PKCS #1 PSS signature scheme. It is defined as follows:497

```
typedef CK ULONG CK_RSA_PKCS_MGF_TYPE;
```

498

499 The following MGFs are defined in PKCS #1. The following table lists the defined functions.

500 *Table 5, PKCS #1 Mask Generation Functions*

Source Identifier	Value
CKG_MGF1_SHA1	0x00000001UL
CKG_MGF1_SHA224	0x00000005UL
CKG_MGF1_SHA256	0x00000002UL
CKG_MGF1_SHA384	0x00000003UL
CKG_MGF1_SHA512	0x00000004UL
CKG_MGF1_SHA3_224	0x00000006UL
CKG_MGF1_SHA3_256	0x00000007UL
CKG_MGF1_SHA3_384	0x00000008UL
CKG_MGF1_SHA3_512	0x00000009UL

501 **CK_RSA_PKCS_MGF_TYPE_PTR** is a pointer to a **CK_RSA_PKCS_MGF_TYPE**.502 ◆ **CK_RSA_PKCS_OAEP_SOURCE_TYPE;
CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR**504 **CK_RSA_PKCS_OAEP_SOURCE_TYPE** is used to indicate the source of the encoding parameter
505 when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:506

```
typedef CK ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;
```

507
508 The following encoding parameter sources are defined in PKCS #1. The following table lists the defined
509 sources along with the corresponding data type for the *pSourceData* field in the
510 **CK_RSA_PKCS_OAEP_PARAMS** structure defined below.

511 *Table 6, PKCS #1 RSA OAEP: Encoding parameter sources*

Source Identifier	Value	Data Type
CKZ_DATA_SPECIFIED	0x00000001UL	Array of CK_BYTE containing the value of the encoding parameter. If the parameter is empty, <i>pSourceData</i> must be NULL and <i>ulSourceDataLen</i> must be zero.

512 **CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR** is a pointer to a
513 **CK_RSA_PKCS_OAEP_SOURCE_TYPE**.

514 ◆ **CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR**

515 **CK_RSA_PKCS_OAEP_PARAMS** is a structure that provides the parameters to the
516 **CKM_RSA_PKCS_OAEP** mechanism. The structure is defined as follows:

```
517     typedef struct CK_RSA_PKCS_OAEP_PARAMS {  
518         CK_MECHANISM_TYPE           hashAlg;  
519         CK_RSA_PKCS_MGF_TYPE        mgf;  
520         CK_RSA_PKCS_OAEP_SOURCE_TYPE source;  
521         CK_VOID_PTR                pSourceData;  
522         CK_ULONG                   ulSourceDataLen;  
523     } CK_RSA_PKCS_OAEP_PARAMS;
```

524
525 The fields of the structure have the following meanings:

526 *hashAlg* *mechanism ID of the message digest algorithm used to calculate*
527 *the digest of the encoding parameter*

528 *mgf* *mask generation function to use on the encoded block*

529 *source* *source of the encoding parameter*

530 *pSourceData* *data used as the input for the encoding parameter source*

531 *ulSourceDataLen* *length of the encoding parameter source input*

532 **CK_RSA_PKCS_OAEP_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_OAEP_PARAMS**.

533 Table 7, Add to following to table 7 (PKCS #1 Mask Generation Functions) in section 2.1..7 (PKCS #1
534 RSA OAEP mechanism parameters)

Source Identifier	Value
CKG_MGF1_SHA3_224	0x00000006UL
CKG_MGF1_SHA3_256	0x00000007UL
CKG_MGF1_SHA3_384	0x00000008UL
CKG_MGF1_SHA3_512	0x00000009UL

535

536 **2.1.8 PKCS #1 RSA OAEP**

537 The PKCS #1 RSA OAEP mechanism, denoted **CKM_RSA_PKCS_OAEP**, is a multi-purpose
538 mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1.
539 It supports single-part encryption and decryption; key wrapping; and key unwrapping.

540 It has a parameter, a **CK_RSA_PKCS_OAEP_PARAMS** structure.

541 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
542 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
543 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
544 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
545 key, except the key length; the application must convey these separately. In particular, the mechanism
546 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
547 to the recovered key during unwrapping; other attributes must be specified in the template.

548 Constraints on key types and the length of the data are summarized in the following table. For encryption
549 and decryption, the input and output data may begin at the same location in memory. In the table, k is the
550 length in bytes of the RSA modulus, and $hLen$ is the output length of the message digest algorithm
551 specified by the *hashAlg* field of the **CK_RSA_PKCS_OAEP_PARAMS** structure.

552 *Table 8, PKCS #1 RSA OAEP: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt ¹	RSA public key	$\leq k-2-2hLen$	k
C_Decrypt ¹	RSA private key	k	$\leq k-2-2hLen$
C_WrapKey	RSA public key	$\leq k-2-2hLen$	k
C_UnwrapKey	RSA private key	k	$\leq k-2-2hLen$

553 ¹ Single-part operations only.

554 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
555 specify the supported range of RSA modulus sizes, in bits.

556 **2.1.9 PKCS #1 RSA PSS mechanism parameters**

557 ◆ **CK_RSA_PKCS_PSS_PARAMS; CK_RSA_PKCS_PSS_PARAMS_PTR**

558 **CK_RSA_PKCS_PSS_PARAMS** is a structure that provides the parameters to the
559 **CKM_RSA_PKCS_PSS** mechanism. The structure is defined as follows:

```
560     typedef struct CK_RSA_PKCS_PSS_PARAMS {
561         CK_MECHANISM_TYPE      hashAlg;
562         CK_RSA_PKCS_MGF_TYPE   mgf;
563         CK ULONG                sLen;
564     } CK_RSA_PKCS_PSS_PARAMS;
```

565

566 The fields of the structure have the following meanings:

567 *hashAlg* *hash algorithm used in the PSS encoding; if the signature*
568 *mechanism does not include message hashing, then this value must*
569 *be the mechanism used by the application to generate the message*
570 *hash; if the signature mechanism includes hashing, then this value*
571 *must match the hash algorithm indicated by the signature*
572 *mechanism*

573 *mgf* *mask generation function to use on the encoded block*

574 *sLen* *length, in bytes, of the salt value used in the PSS encoding; typical*
575 *values are the length of the message hash and zero*

576 **CK_RSA_PKCS_PSS_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_PSS_PARAMS**.

577 **2.1.10 PKCS #1 RSA PSS**

578 The PKCS #1 RSA PSS mechanism, denoted **CKM_RSA_PKCS_PSS**, is a mechanism based on the
579 RSA public-key cryptosystem and the PSS block format defined in PKCS #1. It supports single-part
580 signature generation and verification without message recovery. This mechanism corresponds only to the
581 part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash
582 value on the message to be signed.

583 It has a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must be less than or
584 equal to $k^*-2-hLen$ and *hLen* is the length of the input to the C_Sign or C_Verify function. k^* is the length
585 in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple
586 of 8, in which case k^* is one less than the length in bytes of the RSA modulus.

587 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*
588 is the length in bytes of the RSA.

589 *Table 9, PKCS #1 RSA PSS: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign ¹	RSA private key	<i>hLen</i>	<i>k</i>
C_Verify ¹	RSA public key	<i>hLen, k</i>	N/A

590 ¹ Single-part operations only.

591 ² Data length, signature length.

592 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
593 specify the supported range of RSA modulus sizes, in bits.

594 **2.1.11 ISO/IEC 9796 RSA**

595 The ISO/IEC 9796 RSA mechanism, denoted **CKM_RSA_9796**, is a mechanism for single-part
596 signatures and verification with and without message recovery based on the RSA public-key
597 cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A.

598 This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly,
599 the following transformations are performed:

- 600 • Data is converted between byte and bit string formats by interpreting the most-significant bit of the
601 leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the
602 trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of
603 the data is a multiple of 8).
- 604 • A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to
605 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string
606 as above; it is converted from a byte string to a bit string by converting the byte string as above, and
607 removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

608 This mechanism does not have a parameter.

609 Constraints on key types and the length of input and output data are summarized in the following table.
610 In the table, *k* is the length in bytes of the RSA modulus.

611 Table 10, ISO/IEC 9796 RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Sign ¹	RSA private key	$\leq \lfloor k/2 \rfloor$	k
C_SignRecover	RSA private key	$\leq \lfloor k/2 \rfloor$	k
C_Verify ¹	RSA public key	$\leq \lfloor k/2 \rfloor, k^2$	N/A
C_VerifyRecover	RSA public key	k	$\leq \lfloor k/2 \rfloor$

612 1 Single-part operations only.

613 2 Data length, signature length.

614 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
615 specify the supported range of RSA modulus sizes, in bits.616

2.1.12 X.509 (raw) RSA

617 The X.509 (raw) RSA mechanism, denoted **CKM_RSA_X_509**, is a multi-purpose mechanism based on
618 the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures
619 and verification with and without message recovery; key wrapping; and key unwrapping. All these
620 operations are based on so-called “raw” RSA, as assumed in X.509.621 “Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first,
622 applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first.
623 The input string, considered as an integer, must be less than the modulus; the output string is also less
624 than the modulus.

625 This mechanism does not have a parameter.

626 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
627 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
628 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
629 similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other
630 information about the key; the application must convey these separately, and supply them when
631 unwrapping the key.632 Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism,
633 padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the
634 sequence of plaintext bytes $b_1 b_2 \dots b_n$ ($n \leq k$), Cryptoki forms $P=2^{n-1}b_1+2^{n-2}b_2+\dots+b_n$. This number must
635 be less than the RSA modulus. The k -byte ciphertext (k is the length in bytes of the RSA modulus) is
636 produced by raising P to the RSA public exponent modulo the RSA modulus. Decryption of a k -byte
637 ciphertext C is accomplished by raising C to the RSA private exponent modulo the RSA modulus, and
638 returning the resulting value as a sequence of exactly k bytes. If the resulting plaintext is to be used to
639 produce an unwrapped key, then however many bytes are specified in the template for the length of the
640 key are taken from the end of this sequence of bytes.641 Technically, the above procedures may differ very slightly from certain details of what is specified in
642 X.509.643 Executing cryptographic operations using this mechanism can result in the error returns
644 **CKR_DATA_INVALID** (if plaintext is supplied which has the same length as the RSA modulus and is
645 numerically at least as large as the modulus) and **CKR_ENCRYPTED_DATA_INVALID** (if ciphertext is
646 supplied which has the same length as the RSA modulus and is numerically at least as large as the
647 modulus).648 Constraints on key types and the length of input and output data are summarized in the following table.
649 In the table, k is the length in bytes of the RSA modulus.

650 *Table 11, X.509 (Raw) RSA: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt ¹	RSA public key	$\leq k$	k
C_Decrypt ¹	RSA private key	k	k
C_Sign ¹	RSA private key	$\leq k$	k
C_SignRecover	RSA private key	$\leq k$	k
C_Verify ¹	RSA public key	$\leq k, k^2$	N/A
C_VerifyRecover	RSA public key	k	k
C_WrapKey	RSA public key	$\leq k$	k
C_UnwrapKey	RSA private key	k	$\leq k$ (specified in template)

651 ¹ Single-part operations only.652 ² Data length, signature length.653 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
654 specify the supported range of RSA modulus sizes, in bits.655 This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC
656 9796 block formats.

657 2.1.13 ANSI X9.31 RSA

658 The ANSI X9.31 RSA mechanism, denoted **CKM_RSA_X9_31**, is a mechanism for single-part signatures
659 and verification without message recovery based on the RSA public-key cryptosystem and the block
660 formats defined in ANSI X9.31.661 This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must
662 be applied by the application.663 This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly,
664 the following transformations are performed:

- Data is converted between byte and bit string formats by interpreting the most-significant bit of the leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of the data is a multiple of 8).
- A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string as above; it is converted from a byte string to a bit string by converting the byte string as above, and removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

673 This mechanism does not have a parameter.

674 Constraints on key types and the length of input and output data are summarized in the following table.
675 In the table, k is the length in bytes of the RSA modulus. For all operations, the k value must be at least
676 128 and a multiple of 32 as specified in ANSI X9.31.677 *Table 12, ANSI X9.31 RSA: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign ¹	RSA private key	$\leq k-2$	k
C_Verify ¹	RSA public key	$\leq k-2, k^2$	N/A

678 ¹ Single-part operations only.679 ² Data length, signature length.

680 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
681 specify the supported range of RSA modulus sizes, in bits.

682 **2.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA- 683 384, SHA-512, RIPE-MD 128 or RIPE-MD 160**

684 The PKCS #1 v1.5 RSA signature with MD2 mechanism, denoted **CKM_MD2_RSA_PKCS**, performs
685 single- and multiple-part digital signatures and verification operations without message recovery. The
686 operations performed are as described initially in PKCS #1 v1.5 with the object identifier
687 *md2WithRSAEncryption*, and as in the scheme RSASSA-PKCS1-v1_5 in the current version of PKCS #1,
688 where the underlying hash function is MD2.

689 Similarly, the PKCS #1 v1.5 RSA signature with MD5 mechanism, denoted **CKM_MD5_RSA_PKCS**,
690 performs the same operations described in PKCS #1 with the object identifier *md5WithRSAEncryption*.
691 The PKCS #1 v1.5 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_PKCS**, performs
692 the same operations, except that it uses the hash function SHA-1 with object identifier
693 *sha1WithRSAEncryption*.

694 Likewise, the PKCS #1 v1.5 RSA signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted
695 **CKM_SHA256_RSA_PKCS**, **CKM_SHA384_RSA_PKCS**, and **CKM_SHA512_RSA_PKCS** respectively,
696 perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions with the object
697 identifiers *sha256WithRSAEncryption*, *sha384WithRSAEncryption* and *sha512WithRSAEncryption*
698 respectively.

699 The PKCS #1 v1.5 RSA signature with RIPEMD-128 or RIPEMD-160, denoted
700 **CKM_RIPEMD128_RSA_PKCS** and **CKM_RIPEMD160_RSA_PKCS** respectively, perform the same
701 operations using the RIPE-MD 128 and RIPE-MD 160 hash functions.

702 None of these mechanisms has a parameter.

703 Constraints on key types and the length of the data for these mechanisms are summarized in the
704 following table. In the table, *k* is the length in bytes of the RSA modulus. For the PKCS #1 v1.5 RSA
705 signature with MD2 and PKCS #1 v1.5 RSA signature with MD5 mechanisms, *k* must be at least 27; for
706 the PKCS #1 v1.5 RSA signature with SHA-1 mechanism, *k* must be at least 31, and so on for other
707 underlying hash functions, where the minimum is always 11 bytes more than the length of the hash value.

708 *Table 13, PKCS #1 v1.5 RSA Signatures with Various Hash Functions: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Sign	RSA private key	any	<i>k</i>	block type 01
C_Verify	RSA public key	any, <i>k</i> ²	N/A	block type 01

709 ² Data length, signature length.

710 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
711 structure specify the supported range of RSA modulus sizes, in bits.

712 **2.1.15 PKCS #1 v1.5 RSA signature with SHA-224**

713 The PKCS #1 v1.5 RSA signature with SHA-224 mechanism, denoted **CKM_SHA224_RSA_PKCS**,
714 performs similarly as the other **CKM_SHAX_RSA_PKCS** mechanisms but uses the SHA-224 hash
715 function.

716 **2.1.16 PKCS #1 RSA PSS signature with SHA-224**

717 The PKCS #1 RSA PSS signature with SHA-224 mechanism, denoted **CKM_SHA224_RSA_PKCS_PSS**,
718 performs similarly as the other **CKM_SHAX_RSA_PSS** mechanisms but uses the SHA-224 hash
719 function.

2.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512

The PKCS #1 RSA PSS signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_PKCS_PSS**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in PKCS #1 with the object identifier id-RSASSA-PSS, i.e., as in the scheme RSASSA-PSS in PKCS #1 where the underlying hash function is SHA-1.

The PKCS #1 RSA PSS signature with SHA-256, SHA-384, and SHA-512 mechanisms, denoted **CKM_SHA256_RSA_PKCS_PSS**, **CKM_SHA384_RSA_PKCS_PSS**, and **CKM_SHA512_RSA_PKCS_PSS** respectively, perform the same operations using the SHA-256, SHA-384 and SHA-512 hash functions.

The mechanisms have a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must be less than or equal to $k^* - hLen$ where *hLen* is the length in bytes of the hash value. k^* is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple of 8, in which case k^* is one less than the length in bytes of the RSA modulus.

Constraints on key types and the length of the data are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus.

Table 14, PKCS #1 RSA PSS Signatures with Various Hash Functions: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	RSA private key	any	<i>k</i>
C_Verify	RSA public key	any, k^2	N/A

2 Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of RSA modulus sizes, in bits.

2.1.18 PKCS #1 v1.5 RSA signature with SHA3

The PKCS #1 v1.5 RSA signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted **CKM_SHA3_224_RSA_PKCS**, **CKM_SHA3_256_RSA_PKCS**, **CKM_SHA3_384_RSA_PKCS**, and **CKM_SHA3_512_RSA_PKCS** respectively, performs similarly as the other **CKM_SHAX_RSA_PKCS** mechanisms but uses the corresponding SHA3 hash functions.

2.1.19 PKCS #1 RSA PSS signature with SHA3

The PKCS #1 RSA PSS signature with SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanisms, denoted **CKM_SHA3_224_RSA_PSS**, **CKM_SHA3_256_RSA_PSS**, **CKM_SHA3_384_RSA_PSS**, and **CKM_SHA3_512_RSA_PSS** respectively, performs similarly as the other **CKM_SHAX_RSA_PSS** mechanisms but uses the corresponding SHA-3 hash functions.

2.1.20 ANSI X9.31 RSA signature with SHA-1

The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_X9_31**, performs single- and multiple-part digital signatures and verification operations without message recovery. The operations performed are as described in ANSI X9.31.

This mechanism does not have a parameter.

Constraints on key types and the length of the data for these mechanisms are summarized in the following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value must be at least 128 and a multiple of 32 as specified in ANSI X9.31.

758 Table 15, ANSI X9.31 RSA Signatures with SHA-1: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	RSA private key	any	k
C_Verify	RSA public key	any, k^2	N/A

759 2 Data length, signature length.

760 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
761 structure specify the supported range of RSA modulus sizes, in bits.

762 2.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA

763 The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS TPM_1_1**, is a
764 multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in
765 PKCS #1 v1.5, with additional formatting rules defined in TCPA TPM Specification Version 1.1b.
766 Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports
767 single-part encryption and decryption; key wrapping; and key unwrapping.

768 This mechanism does not have a parameter. It differs from the standard PKCS#1 v1.5 RSA encryption
769 mechanism in that the plaintext is wrapped in a TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM
770 1.2) structure before being submitted to the PKCS#1 v1.5 encryption process. On encryption, the version
771 field of the TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM 1.2) structure must contain 0x01, 0x01,
772 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be accepted.

773 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
774 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
775 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
776 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
777 key, except the key length; the application must convey these separately. In particular, the mechanism
778 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
779 to the recovered key during unwrapping; other attributes must be specified in the template.

780 Constraints on key types and the length of the data are summarized in the following table. For encryption
781 and decryption, the input and output data may begin at the same location in memory. In the table, k is the
782 length in bytes of the RSA modulus.

783 Table 16, TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt ¹	RSA public key	$\leq k-11-5$	k
C_Decrypt ¹	RSA private key	k	$\leq k-11-5$
C_WrapKey	RSA public key	$\leq k-11-5$	k
C_UnwrapKey	RSA private key	k	$\leq k-11-5$

784 1 Single-part operations only.

785

786 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
787 specify the supported range of RSA modulus sizes, in bits.

788 2.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP

789 The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted
790 **CKM_RSA_PKCS_OAEP TPM_1_1**, is a multi-purpose mechanism based on the RSA public-key
791 cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCPA
792 TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM
793 Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key
794 unwrapping.

795 This mechanism does not have a parameter. It differs from the standard PKCS#1 OAEP RSA encryption
796 mechanism in that the plaintext is wrapped in a TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM
797 1.2) structure before being submitted to the encryption process and that all of the values of the
798 parameters that are passed to a standard CKM_RSA_PKCS_OAEP operation are fixed. On encryption,
799 the version field of the TCPA_BOUND_DATA (TPM_BOUND_DATA for TPM 1.2) structure must contain
800 0x01, 0x01, 0x00, 0x00. On decryption, any structure of the form 0x01, 0x01, 0xXX, 0xYY may be
801 accepted.

802 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
803 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
804 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
805 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
806 key, except the key length; the application must convey these separately. In particular, the mechanism
807 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
808 to the recovered key during unwrapping; other attributes must be specified in the template.

809 Constraints on key types and the length of the data are summarized in the following table. For encryption
810 and decryption, the input and output data may begin at the same location in memory. In the table, k is the
811 length in bytes of the RSA modulus.

812 *Table 17, TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP: Key And Data Length*

Function	Key type	Input length	Output length
C_Encrypt ¹	RSA public key	$\leq k-2-40-5$	k
C_Decrypt ¹	RSA private key	k	$\leq k-2-40-5$
C_WrapKey	RSA public key	$\leq k-2-40-5$	k
C_UnwrapKey	RSA private key	k	$\leq k-2-40-5$

813 ¹ Single-part operations only.

814 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
815 specify the supported range of RSA modulus sizes, in bits.

816 2.1.23 RSA AES KEY WRAP

817 The RSA AES key wrap mechanism, denoted **CKM_RSA_AES_KEY_WRAP**, is a mechanism based on
818 the RSA public-key cryptosystem and the AES key wrap mechanism. It supports single-part key
819 wrapping; and key unwrapping.

820 It has a parameter, a **CK_RSA_AES_KEY_WRAP_PARAMS** structure.

821 The mechanism can wrap and unwrap a target asymmetric key of any length and type using an RSA
822 key.

- 823 - A temporary AES key is used for wrapping the target key using
824 **CKM_AES_KEY_WRAP_KWP** mechanism.
825 - The temporary AES key is wrapped with the wrapping RSA key using
826 **CKM_RSA_PKCS_OAEP** mechanism.

827 For wrapping, the mechanism -

- 829 • Generates a temporary random AES key of *ulAESKeyBits* length. This key is not accessible to
830 the user - no handle is returned.
- 831 • Wraps the AES key with the wrapping RSA key using **CKM_RSA_PKCS_OAEP** with parameters
832 of *OAEPPParams*.
- 833 • Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP** ([AES
834 KEYWRAP] section 6.3).
- 835 • Zeroizes the temporary AES key

- 836 • Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped
 837 AES key, and the second is the wrapped target key.

838
 839 The recommended format for an asymmetric target key being wrapped is as a PKCS8
 840 PrivateKeyInfo

841
 842 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the
 843 object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown

844
 845 For unwrapping, the mechanism -

- 846 • Splits the input into two parts. The first is the wrapped AES key, and the second is the wrapped
 847 target key. The length of the first part is equal to the length of the unwrapping RSA key.
- 848 • Un-wraps the temporary AES key from the first part with the private RSA key using
 849 **CKM_RSA_PKCS_OAEP** with parameters of *OAEPPParams*.
- 850 • Un-wraps the target key from the second part with the temporary AES key using
 851 **CKM_AES_KEY_WRAP_KWP** ([AES KEYWRAP] section 6.3).
- 852 • Zeroizes the temporary AES key.
- 853 • Returns the handle to the newly unwrapped target key.

854 Table 18, CKM_RSA_AES_KEY_WRAP Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_RSA_AES_KEY_WRAP						✓	

¹SR = SignRecover, VR = VerifyRecover

2.1.24 RSA AES KEY WRAP mechanism parameters

- 855 ◆ **CK_RSA_AES_KEY_WRAP_PARAMS; CK_RSA_AES_KEY_WRAP_PARAMS_PTR**

856 **CK_RSA_AES_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
 857 **CKM_RSA_AES_KEY_WRAP** mechanism. It is defined as follows:

```
859       typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
860           CK ULONG                                      ulAESKeyBits;
861           CK RSA_PKCS_OAEP_PARAMS_PTR              pOAEPPParams;
862       } CK RSA AES KEY_WRAP_PARAMS;
```

863

864 The fields of the structure have the following meanings:

865 length of the temporary AES key in bits. Can be only 128, 192 or
 866 256.

867 pointer to the parameters of the temporary AES key wrapping. See
 868 also the description of PKCS #1 RSA OAEP mechanism
 869 parameters.

870 CK_RSA_AES_KEY_WRAP_PARAMS_PTR is a pointer to a CK_RSA_AES_KEY_WRAP_PARAMS.

871 2.1.25 FIPS 186-4

872 When CKM_RSA_PKCS is operated in FIPS mode, the length of the modulus SHALL only be 1024,
873 2048, or 3072 bits.

874 2.2 DSA

875 *Table 19, DSA Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DSA_KEY_PAIR_GEN					✓		
CKM_DSA_PARAMETER_GEN					✓		
CKM_DSA_PROBABALISTIC_PARAMETER_GEN					✓		
CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN					✓		
CKM_DSA_FIPS_G_GEN					✓		
CKM_DSA		✓ ²					
CKM_DSA_SHA1		✓					
CKM_DSA_SHA224		✓					
CKM_DSA_SHA256		✓					
CKM_DSA_SHA384		✓					
CKM_DSA_SHA512		✓					
CKM_DSA_SHA3_224		✓					
CKM_DSA_SHA3_256		✓					
CKM_DSA_SHA3_384		✓					
CKM_DSA_SHA3_512		✓					

876 2.2.1 Definitions

877 This section defines the key type “CKK_DSA” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
878 attribute of DSA key objects.

879 Mechanisms:

880 CKM_DSA_KEY_PAIR_GEN

881 CKM_DSA

882 CKM_DSA_SHA1

883 CKM_DSA_SHA224

884 CKM_DSA_SHA256

885 CKM_DSA_SHA384

886 CKM_DSA_SHA512

887 CKM_DSA_SHA3_224

888 CKM_DSA_SHA3_256

889 CKM_DSA_SHA3_384

```

890      CKM_DSA_SHA3_512
891      CKM_DSA_PARAMETER_GEN
892      CKM_DSA_PROBABLISTIC_PARAMETER_GEN
893      CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN
894      CKM_DSA_FIPS_G_GEN
895

```

896 ◆ CK_DSA_PARAMETER_GEN_PARAM

897 CK_DSA_PARAMETER_GEN_PARAM is a structure which provides and returns parameters for the
 898 NIST FIPS 186-4 parameter generating algorithms.

899

```

900     typedef struct CK_DSA_PARAMETER_GEN_PARAM {
901         CK_MECHANISM_TYPE    hash;
902         CK_BYTE_PTR          pSeed;
903         CK ULONG              ulSeedLen;
904         CK ULONG              ulIndex;
905     } CK_DSA_PARAMETER_GEN_PARAM;
906

```

907 The fields of the structure have the following meanings:

908	<i>hash</i>	<i>Mechanism value for the base hash used in PQG generation, Valid values are CKM_SHA1, CKM_SHA224, CKM_SHA256, CKM_SHA384, CKM_SHA512.</i>
911	<i>pSeed</i>	<i>Seed value used to generate PQ and G. This value is returned by CKM_DSA_PROBABLISTIC_PARAMETER_GEN, CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN, and passed into CKM_DSA_FIPS_G_GEN.</i>
915	<i>ulSeedLen</i>	<i>Length of seed value.</i>
916	<i>ulIndex</i>	<i>Index value for generating G. Input for CKM_DSA_FIPS_G_GEN. Ignored by CKM_DSA_PROBABALISTIC_PARAMETER_GEN and CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN.</i>

919 2.2.2 DSA public key objects

920 DSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_DSA**) hold DSA public keys.
 921 The following table defines the DSA public key object attributes, in addition to the common attributes
 922 defined for this object class:

923 *Table 20, DSA Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME ^{1,3}	Big integer	Prime p (512 to 3072 bits, in steps of 64 bits)
CKA_SUBPRIME ^{1,3}	Big integer	Subprime q (160, 224 bits, or 256 bits)
CKA_BASE ^{1,3}	Big integer	Base g
CKA_VALUE ^{1,4}	Big integer	Public value y

924 - Refer to [PKCS11-Base] table 11 for footnotes

925 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
 926 parameters”. See FIPS PUB 186-4 for more information on DSA keys.

```

927 The following is a sample template for creating a DSA public key object:
928     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
929     CK_KEY_TYPE keyType = CKK_DSA;
930     CK_UTF8CHAR label[] = "A DSA public key object";
931     CK_BYTE prime[] = { ... };
932     CK_BYTE subprime[] = { ... };
933     CK_BYTE base[] = { ... };
934     CK_BYTE value[] = { ... };
935     CK_BBOOL true = CK_TRUE;
936     CK_ATTRIBUTE template[] = {
937         {CKA_CLASS, &class, sizeof(class)},
938         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
939         {CKA_TOKEN, &true, sizeof(true)},
940         {CKA_LABEL, label, sizeof(label)-1},
941         {CKA_PRIME, prime, sizeof(prime)},
942         {CKA_SUBPRIME, subprime, sizeof(subprime)},
943         {CKA_BASE, base, sizeof(base)},
944         {CKA_VALUE, value, sizeof(value)}}
945     } ;
946

```

2.2.3 DSA Key Restrictions

947 FIPS PUB 186-4 specifies permitted combinations of prime and sub-prime lengths. They are:

- 949 • Prime: 1024 bits, Subprime: 160
- 950 • Prime: 2048 bits, Subprime: 224
- 951 • Prime: 2048 bits, Subprime: 256
- 952 • Prime: 3072 bits, Subprime: 256

953 Earlier versions of FIPS 186 permitted smaller prime lengths, and those are included here for backwards
954 compatibility. An implementation that is compliant to FIPS 186-4 does not permit the use of primes of
955 any length less than 1024 bits.

2.2.4 DSA private key objects

957 DSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_DSA**) hold DSA private keys.
958 The following table defines the DSA private key object attributes, in addition to the common attributes
959 defined for this object class:

960 *Table 21, DSA Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME ^{1,4,6}	Big integer	Prime p (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME ^{1,4,6}	Big integer	Subprime q (160 bits, 224 bits, or 256 bits)
CKA_BASE ^{1,4,6}	Big integer	Base g
CKA_VALUE ^{1,4,6,7}	Big integer	Private value x

961 - Refer to [PKCS11-Base] table 11 for footnotes

962 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
963 parameters”. See FIPS PUB 186-4 for more information on DSA keys.

964 Note that when generating a DSA private key, the DSA domain parameters are *not* specified in the key's
965 template. This is because DSA private keys are only generated as part of a DSA key pair, and the DSA
966 domain parameters for the pair are specified in the template for the DSA public key.

967 The following is a sample template for creating a DSA private key object:

```
968 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
969 CK_KEY_TYPE keyType = CKK_DSA;  
970 CK_UTF8CHAR label[] = "A DSA private key object";  
971 CK_BYTE subject[] = {...};  
972 CK_BYTE id[] = {123};  
973 CK_BYTE prime[] = {...};  
974 CK_BYTE subprime[] = {...};  
975 CK_BYTE base[] = {...};  
976 CK_BYTE value[] = {...};  
977 CK_BBOOL true = CK_TRUE;  
978 CK_ATTRIBUTE template[] = {  
979     {CKA_CLASS, &class, sizeof(class)},  
980     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
981     {CKA_TOKEN, &true, sizeof(true)},  
982     {CKA_LABEL, label, sizeof(label)-1},  
983     {CKA SUBJECT, subject, sizeof(subject)},  
984     {CKA_ID, id, sizeof(id)},  
985     {CKA_SENSITIVE, &true, sizeof(true)},  
986     {CKA_SIGN, &true, sizeof(true)},  
987     {CKA_PRIME, prime, sizeof(prime)},  
988     {CKA_SUBPRIME, subprime, sizeof(subprime)},  
989     {CKA_BASE, base, sizeof(base)},  
990     {CKA_VALUE, value, sizeof(value)}  
991 };
```

992 2.2.5 DSA domain parameter objects

993 DSA domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_DSA**) hold
994 DSA domain parameters. The following table defines the DSA domain parameter object attributes, in
995 addition to the common attributes defined for this object class:

996 *Table 22, DSA Domain Parameter Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME ^{1,4}	Big integer	Prime p (512 to 1024 bits, in steps of 64 bits)
CKA_SUBPRIME ^{1,4}	Big integer	Subprime q (160 bits, 224 bits, or 256 bits)
CKA_BASE ^{1,4}	Big integer	Base g
CKA_PRIME_BITS ^{2,3}	CK ULONG	Length of the prime value.

997 - Refer to [PKCS11-Base] table 11 for footnotes

998 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the "DSA domain
999 parameters". See FIPS PUB 186-4 for more information on DSA domain parameters.

1000 To ensure backwards compatibility, if **CKA_SUBPRIME_BITS** is not specified for a call to
1001 **C_GenerateKey**, it takes on a default based on the value of **CKA_PRIME_BITS** as follows:

- 1002 • If **CKA_PRIME_BITS** is less than or equal to 1024 then **CKA_SUBPRIME_BITS** shall be 160 bits
1003 • If **CKA_PRIME_BITS** equals 2048 then **CKA_SUBPRIME_BITS** shall be 224 bits

- 1004 • If **CKA_PRIME_BITS** equals 3072 then **CKA_SUBPRIME_BITS** shall be 256 bits

1005
1006 The following is a sample template for creating a DSA domain parameter object:

```
1007       CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
1008       CK_KEY_TYPE keyType = CKK_DSA;
1009       CK_UTF8CHAR label[] = "A DSA domain parameter object";
1010       CK_BYTE prime[] = {...};
1011       CK_BYTE subprime[] = {...};
1012       CK_BYTE base[] = {...};
1013       CK_BBOOL true = CK_TRUE;
1014       CK_ATTRIBUTE template[] = {
1015           {CKA_CLASS, &class, sizeof(class)},
1016           {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
1017           {CKA_TOKEN, &true, sizeof(true)},
1018           {CKA_LABEL, label, sizeof(label)-1},
1019           {CKA_PRIME, prime, sizeof(prime)},
1020           {CKA_SUBPRIME, subprime, sizeof(subprime)},
1021           {CKA_BASE, base, sizeof(base)}},
1022     };
```

1023 2.2.6 DSA key pair generation

1024 The DSA key pair generation mechanism, denoted **CKM_DSA_KEY_PAIR_GEN**, is a key pair generation
1025 mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

1026 This mechanism does not have a parameter.

1027 The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as
1028 specified in the **CKA_PRIME**, **CKA_SUBPRIME**, and **CKA_BASE** attributes of the template for the public
1029 key.

1030 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
1031 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**, **CKA_BASE**, and
1032 **CKA_VALUE** attributes to the new private key. Other attributes supported by the DSA public and private
1033 key types (specifically, the flags indicating which functions the keys support) may also be specified in the
1034 templates for the keys, or else are assigned default initial values.

1035 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1036 specify the supported range of DSA prime sizes, in bits.

1037 2.2.7 DSA domain parameter generation

1038 The DSA domain parameter generation mechanism, denoted **CKM_DSA_PARAMETER_GEN**, is a
1039 domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB
1040 186-2.

1041 This mechanism does not have a parameter.

1042 The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in
1043 the **CKA_PRIME_BITS** attribute of the template.

1044 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
1045 **CKA_BASE** and **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the DSA
1046 domain parameter types may also be specified in the template, or else are assigned default initial values.

1047 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1048 specify the supported range of DSA prime sizes, in bits.

1049 2.2.8 DSA probabilistic domain parameter generation

1050 The DSA probabilistic domain parameter generation mechanism, denoted
1051 **CKM_DSA_PROBABILISTIC_PARAMETER_GEN**, is a domain parameter generation mechanism based
1052 on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.1 Generation and
1053 Validation of Probable Primes..
1054 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
1055 returns the seed (pSeed) and the length (ulSeedLen).
1056 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
1057 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
1058 specified in the **CKA_SUBPRIME_BITS** attribute of the template.
1059 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
1060 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
1061 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
1062 template, or else are assigned default initial values.
1063 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1064 specify the supported range of DSA prime sizes, in bits.

1065 2.2.9 DSA Shawe-Taylor domain parameter generation

1066 The DSA Shawe-Taylor domain parameter generation mechanism, denoted
1067 **CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN**, is a domain parameter generation mechanism
1068 based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2
1069 Construction and Validation of Provable Primes p and q.
1070 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
1071 returns the seed (pSeed) and the length (ulSeedLen).
1072 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
1073 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
1074 specified in the **CKA_SUBPRIME_BITS** attribute of the template.
1075 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
1076 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
1077 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
1078 template, or else are assigned default initial values.
1079 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1080 specify the supported range of DSA prime sizes, in bits.

1081 2.2.10 DSA base domain parameter generation

1082 The DSA base domain parameter generation mechanism, denoted **CKM_DSA_FIPS_G_GEN**, is a base
1083 parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4,
1084 section Appendix A.2 Generation of Generator G.
1085 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash the seed
1086 (pSeed) and the length (ulSeedLen) and the index value.
1087 The mechanism generates the DSA base with the domain parameter specified in the **CKA_PRIME** and
1088 **CKA_SUBPRIME** attributes of the template.
1089 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_BASE** attributes to the new
1090 object. Other attributes supported by the DSA domain parameter types may also be specified in the
1091 template, or else are assigned default initial values.
1092 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1093 specify the supported range of DSA prime sizes, in bits.

2.2.11 DSA without hashing

The DSA without hashing mechanism, denoted **CKM_DSA**, is a mechanism for single-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash value.)

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

It does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 23, DSA: Key And Data Length

Function	Key type	Input length	Output length
C_Sign ¹	DSA private key	20, 28, 32, 48, or 64 bits	2*length of subprime
C_Verify ¹	DSA public key	(20, 28, 32, 48, or 64 bits), (2*length of subprime) ²	N/A

¹ Single-part operations only.

² Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of DSA prime sizes, in bits.

2.2.12 DSA with SHA-1

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA1**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. This mechanism computes the entire DSA specification, including the hashing with SHA-1.

For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

This mechanism does not have a parameter.

Constraints on key types and the length of data are summarized in the following table:

Table 24, DSA with SHA-1: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length ²	N/A

² Data length, signature length.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of DSA prime sizes, in bits.

2.2.13 FIPS 186-4

When CKM_DSA is operated in FIPS mode, only the following bit lengths of *p* and *q*, represented by *L* and *N*, SHALL be used:

L = 1024, *N* = 160

1124 L = 2048, N = 224
1125 L = 2048, N = 256
1126 L = 3072, N = 256
1127

1128 2.2.14 DSA with SHA-224

1129 The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA224**, is a mechanism for single- and multiple-
1130 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.
1131 This mechanism computes the entire DSA specification, including the hashing with SHA-224.
1132 For the purposes of this mechanism, a DSA signature is a string of length 2^{subprime} , corresponding to
1133 the concatenation of the DSA values r and s , each represented most-significant byte first.
1134 This mechanism does not have a parameter.
1135 Constraints on key types and the length of data are summarized in the following table:
1136

Table 25, DSA with SHA-244: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2^{subprime} length
C_Verify	DSA public key	any, 2^{subprime} length ²	N/A

1137 ² Data length, signature length.

1138 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1139 specify the supported range of DSA prime sizes, in bits.

1140 2.2.15 DSA with SHA-256

1141 The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA256**, is a mechanism for single- and multiple-
1142 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.
1143 This mechanism computes the entire DSA specification, including the hashing with SHA-256.
1144 For the purposes of this mechanism, a DSA signature is a string of length 2^{subprime} , corresponding to
1145 the concatenation of the DSA values r and s , each represented most-significant byte first.
1146 This mechanism does not have a parameter.
1147 Constraints on key types and the length of data are summarized in the following table:
1148

Table 26, DSA with SHA-256: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2^{subprime} length
C_Verify	DSA public key	any, 2^{subprime} length ²	N/A

1149 ² Data length, signature length.

1150 2.2.16 DSA with SHA-384

1151 The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA384**, is a mechanism for single- and multiple-
1152 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.
1153 This mechanism computes the entire DSA specification, including the hashing with SHA-384.

1154 For the purposes of this mechanism, a DSA signature is a string of length 2^{*}subprime , corresponding to
 1155 the concatenation of the DSA values r and s , each represented most-significant byte first.
 1156 This mechanism does not have a parameter.
 1157 Constraints on key types and the length of data are summarized in the following table:
 1158 *Table 27, DSA with SHA-384: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2^{*}subprime length
C_Verify	DSA public key	any, 2^{*}subprime length ²	N/A

1159 ² Data length, signature length.

2.2.17 DSA with SHA-512

1160 The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA512**, is a mechanism for single- and multiple-
 1161 part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4.
 1162 This mechanism computes the entire DSA specification, including the hashing with SHA-512.
 1163 For the purposes of this mechanism, a DSA signature is a string of length 2^{*}subprime , corresponding to
 1164 the concatenation of the DSA values r and s , each represented most-significant byte first.
 1165 This mechanism does not have a parameter.
 1166 Constraints on key types and the length of data are summarized in the following table:
 1167 *Table 28, DSA with SHA-512: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2^{*}subprime length
C_Verify	DSA public key	any, 2^{*}subprime length ²	N/A

1168 ² Data length, signature length.

2.2.18 DSA with SHA3-224

1169 The DSA with SHA3-224 mechanism, denoted **CKM_DSA_SHA3_224**, is a mechanism for single- and multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-224.
 1170 For the purposes of this mechanism, a DSA signature is a string of length 2^{*}subprime , corresponding to
 1171 the concatenation of the DSA values r and s , each represented most-significant byte first.
 1172 This mechanism does not have a parameter.
 1173 Constraints on key types and the length of data are summarized in the following table:
 1174 *Table 29, DSA with SHA3-224: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2^{*}subprime length
C_Verify	DSA public key	any, 2^{*}subprime length ²	N/A

1175 ² Data length, signature length.

1180 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1181 specify the supported range of DSA prime sizes, in bits.

1182 **2.2.19 DSA with SHA3-256**

1183 The DSA with SHA3-256 mechanism, denoted **CKM_DSA_SHA3_256**, is a mechanism for single- and
1184 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
1185 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-256.

1186 For the purposes of this mechanism, a DSA signature is a string of length $2^{*\text{subprime}}$, corresponding to
1187 the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

1188 This mechanism does not have a parameter.

1189 Constraints on key types and the length of data are summarized in the following table:

1190 *Table 30, DSA with SHA3-256: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2^{*\text{subprime}}$ length
C_Verify	DSA public key	any, $2^{*\text{subprime}}$ length ²	N/A

1191 ² Data length, signature length.

1192 **2.2.20 DSA with SHA3-384**

1193 The DSA with SHA3-384 mechanism, denoted **CKM_DSA_SHA3_384**, is a mechanism for single- and
1194 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
1195 186-4. This mechanism computes the entire DSA specification, including the hashing with SHA3-384.

1196 For the purposes of this mechanism, a DSA signature is a string of length $2^{*\text{subprime}}$, corresponding to
1197 the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

1198 This mechanism does not have a parameter.

1199 Constraints on key types and the length of data are summarized in the following table:

1200 *Table 31, DSA with SHA3-384: Key And Data Length*

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2^{*\text{subprime}}$ length
C_Verify	DSA public key	any, $2^{*\text{subprime}}$ length ²	N/A

1201 ² Data length, signature length.

1202 **2.2.21 DSA with SHA3-512**

1203 The DSA with SHA3-512 mechanism, denoted **CKM_DSA_SHA3-512**, is a mechanism for single- and
1204 multiple-part signatures and verification based on the Digital Signature Algorithm defined in FIPS PUB
1205 186-4. This mechanism computes the entire DSA specification, including the hashing with SH3A-512.

1206 For the purposes of this mechanism, a DSA signature is a string of length $2^{*\text{subprime}}$, corresponding to
1207 the concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

1208 This mechanism does not have a parameter.

1209 Constraints on key types and the length of data are summarized in the following table:

1210 Table 32, DSA with SHA3-512: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	2*subprime length
C_Verify	DSA public key	any, 2*subprime length ²	N/A

1211 ² Data length, signature length.

1212

1213

2.3 Elliptic Curve

1214 The Elliptic Curve (EC) cryptosystem (also related to ECDSA) in this document was originally based on
1215 the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.1216 The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves
1217 were always represented in their Weierstrass form. Since that time, new curves represented in Edwards
1218 form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new
1219 curves, the EC cryptosystem in this document has been extended from the original. Additional key
1220 generation mechanisms have been added as well as an additional signature generation mechanism.

1221

1222 Table 33, Elliptic Curve Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_EC_KEY_PAIR_GEN					✓		
CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS					✓		
CKM_EC_EDWARDS_KEY_PAIR_GEN					✓		
CKM_EC_MONTGOMERY_KEY_PAIR_GEN					✓		
CKM_ECDSA		✓ ²					
CKM_ECDSA_SHA1		✓					
CKM_ECDSA_SHA224		✓					
CKM_ECDSA_SHA256		✓					
CKM_ECDSA_SHA384		✓					
CKM_ECDSA_SHA512		✓					
CKM_ECDSA_SHA3_224		✓					
CKM_ECDSA_SHA3_256		✓					
CKM_ECDSA_SHA3_384		✓					
CKM_ECDSA_SHA3_512		✓					
CKM_EDDSA		✓					
CKM_XEDDSA		✓					
CKM_ECDH1_DERIVE							✓
CKM_ECDH1_COFACTOR_DERIVE							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ECMQV_DERIVE							✓
CKM_ECDH_AES_KEY_WRAP						✓	

1223

1224 Table 34, Mechanism Information Flags

CKF_EC_F_P	0x00100000UL	True if the mechanism can be used with EC domain parameters over F_p
CKF_EC_F_2M	0x00200000UL	True if the mechanism can be used with EC domain parameters over F_{2^m}
CKF_EC_ECPARAMETERS	0x00400000UL	True if the mechanism can be used with EC domain parameters of the choice ecParameters
CKF_EC_OID	0x00800000UL	True if the mechanism can be used with EC domain parameters of the choice old
CKF_EC_UNCOMPRESS	0x01000000UL	True if the mechanism can be used with elliptic curve point uncompressed
CKF_EC_COMPRESS	0x02000000UL	True if the mechanism can be used with elliptic curve point compressed
CKF_EC_CURVENAME	0x04000000UL	True of the mechanism can be used with EC domain parameters of the choice curveName

1225 Note: CKF_EC_NAMEDCURVE is deprecated with PKCS#11 3.00. It is replaced by CKF_EC_OID.

1226 In these standards, there are two different varieties of EC defined:

- 1227 1. EC using a field with an odd prime number of elements (i.e. the finite field F_p).
 1228 2. EC using a field of characteristic two (i.e. the finite field F_{2^m}).

1229 An EC key in Cryptoki contains information about which variety of EC it is suited for. It is preferable that a
 1230 Cryptoki library, which can perform EC mechanisms, be capable of performing operations with the two
 1231 varieties of EC, however this is not required. The **CK_MECHANISM_INFO** structure **CKF_EC_F_P** flag
 1232 identifies a Cryptoki library supporting EC keys over F_p whereas the **CKF_EC_F_2M** flag identifies a
 1233 Cryptoki library supporting EC keys over F_{2^m} . A Cryptoki library that can perform EC mechanisms must
 1234 set either or both of these flags for each EC mechanism.

1235 In these specifications there are also four representation methods to define the domain parameters for an
 1236 EC key. Only the **ecParameters**, the **old** and the **curveName** choices are supported in Cryptoki. The
 1237 **CK_MECHANISM_INFO** structure **CKF_EC_ECPARAMETERS** flag identifies a Cryptoki library
 1238 supporting the **ecParameters** choice whereas the **CKF_EC_OID** flag identifies a Cryptoki library
 1239 supporting the **old** choice, and the **CKF_EC_CURVENAME** flag identifies a Cryptoki library supporting
 1240 the **curveName** choice. A Cryptoki library that can perform EC mechanisms must set the appropriate
 1241 flag(s) for each EC mechanism.

1242 In these specifications, an EC public key (i.e. EC point Q) or the base point G when the **ecParameters**
 1243 choice is used can be represented as an octet string of the uncompressed form or the compressed form.
 1244 The **CK_MECHANISM_INFO** structure **CKF_EC_UNCOMPRESS** flag identifies a Cryptoki library
 1245 supporting the uncompressed form whereas the **CKF_EC_COMPRESS** flag identifies a Cryptoki library
 1246 supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or
 1247 both of these flags for each EC mechanism.

1248 Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation
1249 of domain parameters or one form may encounter difficulties achieving interoperability with other
1250 implementations.
1251 If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the
1252 attempt should fail with the error code CKR_CURVE_NOT_SUPPORTED. If an attempt to create,
1253 generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain
1254 parameters is made, that attempt should fail with the error code CKR_DOMAIN_PARAMS_INVALID. If
1255 an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that
1256 attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT.

1257 **2.3.1 EC Signatures**

1258 For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at
1259 most two times $nLen$ octets, where $nLen$ is the length in octets of the base point order n . The signature
1260 octets correspond to the concatenation of the ECDSA values r and s , both represented as an octet string
1261 of equal length of at most $nLen$ with the most significant byte first. If r and s have different octet length,
1262 the shorter of both must be padded with leading zero octets such that both have the same octet length.
1263 Loosely spoken, the first half of the signature is r and the second half is s . For signatures created by a
1264 token, the resulting signature is always of length $2nLen$. For signatures passed to a token for verification,
1265 the signature may have a shorter length but must be composed as specified before.

1266 If the length of the hash value is larger than the bit length of n , only the leftmost bits of the hash up to the
1267 length of n will be used. Any truncation is done by the token.

1268 Note: For applications, it is recommended to encode the signature as an octet string of length two times
1269 $nLen$ if possible. This ensures that the application works with PKCS#11 modules which have been
1270 implemented based on an older version of this document. Older versions required all signatures to have
1271 length two times $nLen$. It may be impossible to encode the signature with the maximum length of two
1272 times $nLen$ if the application just gets the integer values of r and s (i.e. without leading zeros), but does
1273 not know the base point order n , because r and s can have any value between zero and the base point
1274 order n .

1275 An EdDSA signature is an octet string of even length which is two times $nLen$ octets, where $nLen$ is
1276 calculated as EdDSA parameter b divided by 8. The signature octets correspond to the concatenation of
1277 the EdDSA values R and S as defined in [RFC 8032], both represented as an octet string of equal length
1278 of $nLen$ bytes in little endian order.

1279 **2.3.2 Definitions**

1280 This section defines the key type “CKK_EC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
1281 attribute of key objects.

1282 Note: CKK_ECDSA is deprecated. It is replaced by CKK_EC.

1283 Mechanisms:

1284

1285 CKM_EC_KEY_PAIR_GEN
1286 CKM_EC_EDWARDS_KEY_PAIR_GEN
1287 CKM_EC_MONTGOMERY_KEY_PAIR_GEN
1288 CKM_ECDSA
1289 CKM_ECDSA_SHA1
1290 CKM_ECDSA_SHA224
1291 CKM_ECDSA_SHA256
1292 CKM_ECDSA_SHA384
1293 CKM_ECDSA_SHA512
1294 CKM_ECDSA_SHA3_224

1295 CKM_ECDSA_SHA3_256
1296 CKM_ECDSA_SHA3_384
1297 CKM_ECDSA_SHA3_512
1298 CKM_EDDSA
1299 CKM_XEDDSA
1300 CKM_ECDH1_DERIVE
1301 CKM_ECDH1_COFACTOR_DERIVE
1302 CKM_ECMQV_DERIVE
1303 CKM_ECDH_AES_KEY_WRAP
1304
1305 CKD_NULL
1306 CKD_SHA1_KDF
1307 CKD_SHA224_KDF
1308 CKD_SHA256_KDF
1309 CKD_SHA384_KDF
1310 CKD_SHA512_KDF
1311 CKD_SHA3_224_KDF
1312 CKD_SHA3_256_KDF
1313 CKD_SHA3_384_KDF
1314 CKD_SHA3_512_KDF
1315 CKD_SHA1_KDF_SP800
1316 CKD_SHA224_KDF_SP800
1317 CKD_SHA256_KDF_SP800
1318 CKD_SHA384_KDF_SP800
1319 CKD_SHA512_KDF_SP800
1320 CKD_SHA3_224_KDF_SP800
1321 CKD_SHA3_256_KDF_SP800
1322 CKD_SHA3_384_KDF_SP800
1323 CKD_SHA3_512_KDF_SP800
1324 CKD_BLAKE2B_160_KDF
1325 CKD_BLAKE2B_256_KDF
1326 CKD_BLAKE2B_384_KDF
1327 CKD_BLAKE2B_512_KDF

1328 2.3.3 ECDSA public key objects

1329 EC (also related to ECDSA) public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC**)
1330 hold EC public keys. The following table defines the EC public key object attributes, in addition to the
1331 common attributes defined for this object class:

1332 Table 35, Elliptic Curve Public Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS ^{1,3}	Byte array	DER-encoding of an ANSI X9.62 Parameters value
CKA_EC_POINT ^{1,4}	Byte array	DER-encoding of ANSI X9.62 ECPoint value Q

1333 - Refer to [PKCS11-Base] table 11 for footnotes

1334 Note: CKA_ECDSA_PARAMS is deprecated. It is replaced by CKA_EC_PARAMS.

1335 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
1336 X9.62 as a choice of three parameter representation methods with the following syntax:

```

1337     Parameters ::= CHOICE {
1338         ecParameters    ECParameters,
1339         oid            CURVES.&id({CurveNames}),
1340         implicitlyCA   NULL,
1341         curveName      PrintableString
1342     }

```

1343

1344 This allows detailed specification of all required values using choice **ecParameters**, the use of **oid** as an
1345 object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to
1346 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve
1347 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oid** or
1348 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used
1349 in Cryptoki.

1350 The following is a sample template for creating an EC (ECDSA) public key object:

```

1351     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
1352     CK_KEY_TYPE keyType = CKK_EC;
1353     CK_UTF8CHAR label[] = "An EC public key object";
1354     CK_BYTE ecParams[] = {...};
1355     CK_BYTE ecPoint[] = {...};
1356     CK_BBOOL true = CK_TRUE;
1357     CK_ATTRIBUTE template[] = {
1358         {CKA_CLASS, &class, sizeof(class)},
1359         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
1360         {CKA_TOKEN, &true, sizeof(true)},
1361         {CKA_LABEL, label, sizeof(label)-1},
1362         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
1363         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}}
1364 ;

```

1365

2.3.4 Elliptic curve private key objects

1366 EC (also related to ECDSA) private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC**)
1367 hold EC private keys. See Section 2.3 for more information about EC. The following table defines the EC
1368 private key object attributes, in addition to the common attributes defined for this object class:

1369 Table 36, Elliptic Curve Private Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS ^{1,4,6}	Byte array	DER-encoding of an ANSI X9.62 Parameters value
CKA_VALUE ^{1,4,6,7}	Big integer	ANSI X9.62 private value d

1370 - Refer to [PKCS11-Base] table 11 for footnotes

1371 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
1372 X9.62 as a choice of three parameter representation methods with the following syntax:

```
1373     Parameters ::= CHOICE {
1374         ecParameters    ECParameters,
1375         oid             CURVES.&id({CurveNames}),
1376         implicitlyCA   NULL,
1377         curveName      PrintableString
1378     }
```

1380 This allows detailed specification of all required values using choice **ecParameters**, the use of **oid** as an
1381 object identifier substitute for a particular set of elliptic curve domain parameters, or **implicitlyCA** to
1382 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve
1383 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **oid** or
1384 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used
1385 in Cryptoki. Note that when generating an EC private key, the EC domain parameters are *not* specified in
1386 the key's template. This is because EC private keys are only generated as part of an EC key pair, and
1387 the EC domain parameters for the pair are specified in the template for the EC public key.

1388 The following is a sample template for creating an EC (ECDSA) private key object:

```
1389     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
1390     CK_KEY_TYPE keyType = CKK_EC;
1391     CK_UTF8CHAR label[] = "An EC private key object";
1392     CK_BYTE subject[] = {...};
1393     CK_BYTE id[] = {123};
1394     CK_BYTE ecParams[] = {...};
1395     CK_BYTE value[] = {...};
1396     CK_BBOOL true = CK_TRUE;
1397     CK_ATTRIBUTE template[] = {
1398         {CKA_CLASS, &class, sizeof(class)},
1399         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
1400         {CKA_TOKEN, &true, sizeof(true)},
1401         {CKA_LABEL, label, sizeof(label)-1},
1402         {CKA_SUBJECT, subject, sizeof(subject)},
1403         {CKA_ID, id, sizeof(id)},
1404         {CKA_SENSITIVE, &true, sizeof(true)},
1405         {CKA_DERIVE, &true, sizeof(true)},
1406         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
1407         {CKA_VALUE, value, sizeof(value)}}
1408 };
```

1409 2.3.5 Edwards Elliptic curve public key objects

1410 Edwards EC public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC_EDWARDS**) hold
1411 Edwards EC public keys. The following table defines the Edwards EC public key object attributes, in
1412 addition to the common attributes defined for this object class:

1413 *Table 37, Edwards Elliptic Curve Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_EC_PARAMS ^{1,3}	Byte array	DER-encoding of a Parameters value as defined above
CKA_EC_POINT ^{1,4}	Byte array	DER-encoding of the b-bit public key value in little endian order as defined in RFC 8032

1414 - Refer to [PKCS #11-Base] table 11 for footnotes

1415 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
1416 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
1417 and Montgomery Elliptic curves. The CKA_EC_PARAMS attribute has the following syntax:

```
1418     Parameters ::= CHOICE {  
1419         ecParameters    ECParameters,  
1420         oId             CURVES.&id({CurveNames}),  
1421         implicitlyCA   NULL,  
1422         curveName      PrintableString  
1423     }
```

1424 Edwards EC public keys only support the use of the **curveName** selection to specify a curve name as
1425 defined in [RFC 8032] and the use of the **oId** selection to specify a curve through an EdDSA algorithm as
1426 defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

1427 The following is a sample template for creating an Edwards EC public key object with Edwards25519
1428 being specified as curveName:

```
1429     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
1430     CK_KEY_TYPE keyType = CKK_EC;  
1431     CK_UTF8CHAR label[] = "An Edwards EC public key object";  
1432     CK_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61,  
1433                 0x72, 0x64, 0x73, 0x32, 0x35, 0x35, 0x31, 0x39};  
1434     CK_BYTE ecPoint[] = {...};  
1435     CK_BBOOL true = CK_TRUE;  
1436     CK_ATTRIBUTE template[] = {  
1437         {CKA_CLASS, &class, sizeof(class)},  
1438         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
1439         {CKA_TOKEN, &true, sizeof(true)},  
1440         {CKA_LABEL, label, sizeof(label)-1},  
1441         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},  
1442         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}  
1443     };
```

1444 2.3.6 Edwards Elliptic curve private key objects

1445 Edwards EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC_EDWARDS**)
1446 hold Edwards EC private keys. See Section 2.3 for more information about EC. The following table
1447 defines the Edwards EC private key object attributes, in addition to the common attributes defined for this
1448 object class:

1449 Table 38, Edwards Elliptic Curve Private Key Object Attributes

Attribute	Data type	Meaning
CKA_EC_PARAMS ^{1,4,6}	Byte array	DER-encoding of a Parameters value as defined above
CKA_VALUE ^{1,4,6,7}	Big integer	b-bit private key value in little endian order as defined in RFC 8032

1450 - Refer to [PKCS #11-Base] table 11 for footnotes

1451 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
1452 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
1453 and Montgomery Elliptic curves. The CKA_EC_PARAMS attribute has the following syntax:

```
1454     Parameters ::= CHOICE {  
1455         ecParameters    ECParameters,  
1456         oId             CURVES.&id({CurveNames}),  
1457         implicitlyCA   NULL,  
1458         curveName      PrintableString  
1459     }
```

1460 Edwards EC private keys only support the use of the **curveName** selection to specify a curve name as
1461 defined in [RFC 8032] and the use of the **oId** selection to specify a curve through an EdDSA algorithm as
1462 defined in [RFC 8410]. Note that keys defined by RFC 8032 and RFC 8410 are incompatible.

1463 Note that when generating an Edwards EC private key, the EC domain parameters are *not* specified in
1464 the key's template. This is because Edwards EC private keys are only generated as part of an Edwards
1465 EC key pair, and the EC domain parameters for the pair are specified in the template for the Edwards EC
1466 public key.

1467 The following is a sample template for creating an Edwards EC private key object:

```
1468     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
1469     CK_KEY_TYPE keyType = CKK_EC;  
1470     CK_UTF8CHAR label[] = "An Edwards EC private key object";  
1471     CK_BYTE subject[] = {...};  
1472     CK_BYTE id[] = {123};  
1473     CK_BYTE ecParams[] = {...};  
1474     CK_BYTE value[] = {...};  
1475     CK_BBOOL true = CK_TRUE;  
1476     CK_ATTRIBUTE template[] = {  
1477         {CKA_CLASS, &class, sizeof(class)},  
1478         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
1479         {CKA_TOKEN, &true, sizeof(true)},  
1480         {CKA_LABEL, label, sizeof(label)-1},  
1481         {CKA_SUBJECT, subject, sizeof(subject)},  
1482         {CKA_ID, id, sizeof(id)},  
1483         {CKA_SENSITIVE, &true, sizeof(true)},  
1484         {CKA_DERIVE, &true, sizeof(true)},  
1485         {CKA_VALUE, value, sizeof(value)}  
1486     };
```

1487 2.3.7 Montgomery Elliptic curve public key objects

1488 Montgomery EC public key objects (object class **CKO_PUBLIC_KEY**, key type
1489 **CKK_EC_MONTGOMERY**) hold Montgomery EC public keys. The following table defines the

1490 Montgomery EC public key object attributes, in addition to the common attributes defined for this object
1491 class:

1492 *Table 39, Montgomery Elliptic Curve Public Key Object Attributes*

Attribute	Data type	Meaning
CKA_EC_PARAMS ^{1,3}	Byte array	DER-encoding of a Parameters value as defined above
CKA_EC_POINT ^{1,4}	Byte array	DER-encoding of the public key value in little endian order as defined in RFC 7748

1493 - Refer to [PKCS #11-Base] table 11 for footnotes

1494 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
1495 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
1496 and Montgomery Elliptic curves. The CKA_EC_PARAMS attribute has the following syntax:

```
1497     Parameters ::= CHOICE {  
1498         ecParameters    ECParameters,  
1499         oId              CURVES.&id({CurveNames}),  
1500         implicitlyCA   NULL,  
1501         curveName       PrintableString  
1502     }
```

1503 Montgomery EC public keys only support the use of the **curveName** selection to specify a curve name as
1504 defined in [RFC7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm as
1505 defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

1506 The following is a sample template for creating a Montgomery EC public key object:

```
1507     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
1508     CK_KEY_TYPE keyType = CKK_EC;  
1509     CK_UTF8CHAR label[] = "A Montgomery EC public key object";  
1510     CK_BYTE ecParams[] = {...};  
1511     CK_BYTE ecPoint[] = {...};  
1512     CK_BBOOL true = CK_TRUE;  
1513     CK_ATTRIBUTE template[] = {  
1514         {CKA_CLASS, &class, sizeof(class)},  
1515         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
1516         {CKA_TOKEN, &true, sizeof(true)},  
1517         {CKA_LABEL, label, sizeof(label)-1},  
1518         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},  
1519         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}  
1520     };
```

1521 2.3.8 Montgomery Elliptic curve private key objects

1522 Montgomery EC private key objects (object class **CKO_PRIVATE_KEY**, key type
1523 **CKK_EC_MONTGOMERY**) hold Montgomery EC private keys. See Section 2.3 for more information
1524 about EC. The following table defines the Montgomery EC private key object attributes, in addition to the
1525 common attributes defined for this object class:

Attribute	Data type	Meaning
CKA_EC_PARAMS ^{1,4,6}	Byte array	DER-encoding of a Parameters value as defined above
CKA_VALUE ^{1,4,6,7}	Big integer	Private key value in little endian order as defined in RFC 7748

1527 - Refer to [PKCS #11-Base] table 11 for footnotes

1528 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
 1529 X9.62 as a choice of three parameter representation methods. A 4th choice is added to support Edwards
 1530 and Montgomery Elliptic curves. The CKA_EC_PARAMS attribute has the following syntax:

```
1531   Parameters ::= CHOICE {
1532     ecParameters  ECParameters,
1533     oId           CURVES.&id({CurveNames}),
1534     implicitlyCA NULL,
1535     curveName    PrintableString
1536 }
```

1537 Edwards EC private keys only support the use of the **curveName** selection to specify a curve name as
 1538 defined in [RFC7748] and the use of the **oID** selection to specify a curve through an ECDH algorithm as
 1539 defined in [RFC 8410]. Note that keys defined by RFC 7748 and RFC 8410 are incompatible.

1540 Note that when generating a Montgomery EC private key, the EC domain parameters are *not* specified in
 1541 the key's template. This is because Montgomery EC private keys are only generated as part of a
 1542 Montgomery EC key pair, and the EC domain parameters for the pair are specified in the template for the
 1543 Montgomery EC public key.

1544 The following is a sample template for creating a Montgomery EC private key object:

```
1545   CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
1546   CK_KEY_TYPE keyType = CKK_EC;
1547   CK_UTF8CHAR label[] = "A Montgomery EC private key object";
1548   CK_BYTE subject[] = {...};
1549   CK_BYTE id[] = {123};
1550   CK_BYTE ecParams[] = {...};
1551   CK_BYTE value[] = {...};
1552   CK_BBOOL true = CK_TRUE;
1553   CK_ATTRIBUTE template[] = {
1554     {CKA_CLASS, &class, sizeof(class)},
1555     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
1556     {CKA_TOKEN, &true, sizeof(true)},
1557     {CKA_LABEL, label, sizeof(label)-1},
1558     {CKA_SUBJECT, subject, sizeof(subject)},
1559     {CKA_ID, id, sizeof(id)},
1560     {CKA_SENSITIVE, &true, sizeof(true)},
1561     {CKA_DERIVE, &true, sizeof(true)},
1562     {CKA_VALUE, value, sizeof(value)}
1563   };
```

1564 2.3.9 Elliptic curve key pair generation

1565 The EC (also related to ECDSA) key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN, is
 1566 a key pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.

1567 The EC (also related to ECDSA) key pair generation mechanism, denoted
1568 CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS, is a key pair generation mechanism that uses the method
1569 defined by FIPS 186-4 Appendix B.4.1.
1570 These mechanisms do not have a parameter.
1571 These mechanisms generate EC public/private key pairs with particular EC domain parameters, as
1572 specified in the **CKA_EC_PARAMS** attribute of the template for the public key. Note that this version of
1573 Cryptoki does not include a mechanism for generating these EC domain parameters.
1574 These mechanism contribute the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
1575 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
1576 attributes to the new private key. Other attributes supported by the EC public and private key types
1577 (specifically, the flags indicating which functions the keys support) may also be specified in the templates
1578 for the keys, or else are assigned default initial values.
1579 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1580 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
1581 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between
1582 2^{200} and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary
1583 notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.
1584 Similarly, 2^{300} is a 301-bit number).

1585 2.3.10 Edwards Elliptic curve key pair generation

1586 The Edwards EC key pair generation mechanism, denoted **CKM_EC_EDWARDS_KEY_PAIR_GEN**, is a
1587 key pair generation mechanism for EC keys over curves represented in Edwards form.

1588 This mechanism does not have a parameter.

1589 The mechanism can only generate EC public/private key pairs over the curves edwards25519 and
1590 edwards448 as defined in RFC 8032 or the curves id-Ed25519 and id-Ed448 as defined in RFC 8410.
1591 These curves can only be specified in the **CKA_EC_PARAMS** attribute of the template for the public key
1592 using the **curveName** or the **old** methods. Attempts to generate keys over these curves using any other
1593 EC key pair generation mechanism will fail with **CKR_CURVE_NOT_SUPPORTED**.

1594 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
1595 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
1596 attributes to the new private key. Other attributes supported by the Edwards EC public and private key
1597 types (specifically, the flags indicating which functions the keys support) may also be specified in the
1598 templates for the keys, or else are assigned default initial values.

1599 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1600 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
1601 mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these two
1602 sizes. A Cryptoki implementation may support one or both of these curves and should set the
1603 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

1604 2.3.11 Montgomery Elliptic curve key pair generation

1605 The Montgomery EC key pair generation mechanism, denoted
1606 **CKM_EC_MONTGOMERY_KEY_PAIR_GEN**, is a key pair generation mechanism for EC keys over
1607 curves represented in Montgomery form.

1608 This mechanism does not have a parameter.

1609 The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519
1610 and curve448 as defined in RFC 7748 or the curves id-X25519 and id-X448 as defined in RFC 8410.
1611 These curves can only be specified in the **CKA_EC_PARAMS** attribute of the template for the public key
1612 using the **curveName** or **old** methods. Attempts to generate keys over these curves using any other EC
1613 key pair generation mechanism will fail with **CKR_CURVE_NOT_SUPPORTED**.

1614 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
1615 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**

1616 attributes to the new private key. Other attributes supported by the EC public and private key types
1617 (specifically, the flags indicating which functions the keys support) may also be specified in the templates
1618 for the keys, or else are assigned default initial values.

1619 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1620 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
1621 mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two
1622 sizes. A Cryptoki implementation may support one or both of these curves and should set the
1623 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

1624 **2.3.12 ECDSA without hashing**

1625 Refer section 2.3.1 for signature encoding.

1626 The ECDSA without hashing mechanism, denoted **CKM_ECDSA**, is a mechanism for single-part
1627 signatures and verification for ECDSA. (This mechanism corresponds only to the part of ECDSA that
1628 processes the hash value, which should not be longer than 1024 bits; it does not compute the hash
1629 value.)

1630 This mechanism does not have a parameter.

1631 Constraints on key types and the length of data are summarized in the following table:

1632 *Table 41, ECDSA without hashing: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign ¹	ECDSA private key	any ³	2nLen
C_Verify ¹	ECDSA public key	any ³ , ≤2nLen ²	N/A

1633 ¹ Single-part operations only.

1634 ² Data length, signature length.

1635 ³ Input the entire raw digest. Internally, this will be truncated to the appropriate number of bits.

1636 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1637 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
1638 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between
1639 2²⁰⁰ and 2³⁰⁰ elements (inclusive), then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in
1640 binary notation, the number 2²⁰⁰ consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.
1641 Similarly, 2³⁰⁰ is a 301-bit number).

1642 **2.3.13 ECDSA with hashing**

1643 Refer to section 2.3.1 for signature encoding.

1644 The ECDSA with SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
1645 mechanism, denoted

1646 **CKM_ECDSA_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]**

1647 respectively, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This
1648 mechanism computes the entire ECDSA specification, including the hashing with SHA-1, SHA-224, SHA-
1649 384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively.

1650 This mechanism does not have a parameter.

1651 Constraints on key types and the length of data are summarized in the following table:

1652 *Table 42, ECDSA with hashing: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign	ECDSA private key	any	2nLen
C_Verify	ECDSA public key	any, ≤2nLen ²	N/A

1653 ² Data length, signature length.

1654 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 1655 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
 1656 example, if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between
 1657 2^{200} and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary
 1658 notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number.
 1659 Similarly, 2^{300} is a 301-bit number).

1660 2.3.14 EdDSA

1661 The EdDSA mechanism, denoted **CKM_EDDSA**, is a mechanism for single-part and multipart signatures
 1662 and verification for EdDSA. This mechanism implements the five EdDSA signature schemes defined in
 1663 RFC 8032 and RFC 8410.

1664 For curves according to RFC 8032, this mechanism has an optional parameter, a **CK_EDDSA_PARAMS**
 1665 structure. The absence or presence of the parameter as well as its content is used to identify which
 1666 signature scheme is to be used. Table 32 enumerates the five signature schemes defined in RFC 8032
 1667 and all supported permutations of the mechanism parameter and its content.

1668 *Table 43, Mapping to RFC 8032 Signature Schemes*

Signature Scheme	Mechanism Param	phFlag	Context Data
Ed25519	<i>Not Required</i>	N/A	N/A
Ed25519ctx	<i>Required</i>	False	Optional
Ed25519ph	<i>Required</i>	True	Optional
Ed448	<i>Required</i>	False	Optional
Ed448ph	<i>Required</i>	True	Optional

1669 For curves according to RFC 8410, the mechanism is implicitly given by the curve, which is EdDSA in
 1670 pure mode.

1671 Constraints on key types and the length of data are summarized in the following table:

1672 *Table 44, EdDSA: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign	<i>CKK_EC_EDWARDS private key</i>	any	$2bLen$
C_Verify	<i>CKK_EC_EDWARDS public key</i>	any, $\leq 2bLen^2$	N/A

1673 ² Data length, signature length.

1674 Note that for EdDSA in pure mode, Ed25519 and Ed448 the data must be processed twice. Therefore, a
 1675 token might need to cache all the data, especially when used with C_SignUpdate/C_VerifyUpdate. If
 1676 tokens are unable to do so they can return CKM_TOKEN_RESOURCE_EXCEEDED.

1677 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 1678 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
 1679 mechanism, the only allowed values are 255 and 448 as RFC 8032 and RFC 8410 only define curves of
 1680 these two sizes. A Cryptoki implementation may support one or both of these curves and should set the
 1681 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

1682 2.3.15 XEdDSA

1683 The XEdDSA mechanism, denoted **CKM_XEDDSA**, is a mechanism for single-part signatures and
 1684 verification for XEdDSA. This mechanism implements the XEdDSA signature scheme defined in
 1685 [XEDDSA]. CKM_XEDDSA operates on CKK_EC_MONTGOMERY type EC keys, which allows these

1686 keys to be used both for signing/verification and for Diffie-Hellman style key-exchanges. This double use
1687 is necessary for the Extended Triple Diffie-Hellman where the long-term identity key is used to sign short-
1688 term keys and also contributes to the DH key-exchange.

1689 This mechanism has a parameter, a **CK_XEDDSA_PARAMS** structure.

1690 *Table 45, XEdDSA: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign ¹	CKK_EC_MONTGOMERY private	any ³	2b
C_Verify ¹	CKK_EC_MONTGOMERY public	any ³ , ≤2b ²	N/A

1691 ² Data length, signature length.

1692 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
1693 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
1694 mechanism, the only allowed values are 255 and 448 as [XEDDSA] only defines curves of these two
1695 sizes. A Cryptoki implementation may support one or both of these curves and should set the
1696 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

1697 2.3.16 EC mechanism parameters

1698 ♦ **CK_EDDSA_PARAMS, CK_EDDSA_PARAMS_PTR**

1699 **CK_EDDSA_PARAMS** is a structure that provides the parameters for the **CKM_EDDSA** signature
1700 mechanism. The structure is defined as follows:

```
1701     typedef struct CK_EDDSA_PARAMS {  
1702         CK_BBOOL      phFlag;  
1703         CK ULONG      ulContextDataLen;  
1704         CK_BYTE_PTR   pContextData;  
1705     } CK_EDDSA_PARAMS;
```

1706

1707 The fields of the structure have the following meanings:

1708 *phFlag* a Boolean value which indicates if Prehashed variant of EdDSA should
1709 be used

1710 *ulContextDataLen* the length in bytes of the context data where 0 <= *ulContextDataLen* <=
1711 255.

1712 *pContextData* context data shared between the signer and verifier

1713 **CK_EDDSA_PARAMS_PTR** is a pointer to a **CK_EDDSA_PARAMS**.

1714

1715 ♦ **CK_XEDDSA_PARAMS, CK_XEDDSA_PARAMS_PTR**

1716 **CK_XEDDSA_PARAMS** is a structure that provides the parameters for the **CKM_XEDDSA** signature
1717 mechanism. The structure is defined as follows:

```

1718     typedef struct CK_XEDDSA_PARAMS {
1719         CK_XEDDSA_HASH_TYPE hash;
1720     } CK_XEDDSA_PARAMS;
1721
1722     The fields of the structure have the following meanings:
1723             hash      a Hash mechanism to be used by the mechanism.
1724
1725     CK_XEDDSA_PARAMS_PTR is a pointer to a CK_XEDDSA_PARAMS.
1726
1727     ♦ CK_XEDDSA_HASH_TYPE, CK_XEDDSA_HASH_TYPE_PTR
1728
1729     CK_XEDDSA_HASH_TYPE is used to indicate the hash function used in XEDDSA. It is defined as
1730 follows:
1731
1732     The following table lists the defined functions.
1733
1734     Table 46, EC: Key Derivation Functions

```

Source Identifier
CKM_BLAKE2B_256
CKM_BLAKE2B_512
CKM_SHA3_256
CKM_SHA3_512
CKM_SHA256
CKM_SHA512

```

1735
1736     CK_XEDDSA_HASH_TYPE_PTR is a pointer to a CK_XEDDSA_HASH_TYPE.
1737
1738     ♦ CK_EC_KDF_TYPE, CK_EC_KDF_TYPE_PTR
1739
1740     CK_EC_KDF_TYPE is used to indicate the Key Derivation Function (KDF) applied to derive keying data
1741 from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is
1742 defined as follows:
1743
1744     The following table lists the defined functions.
1745
1746     Table 47, EC: Key Derivation Functions

```

Source Identifier
CKD_NULL
CKD_SHA1_KDF
CKD_SHA224_KDF
CKD_SHA256_KDF
CKD_SHA384_KDF
CKD_SHA512_KDF
CKD_SHA3_224_KDF

CKD_SHA3_256_KDF
CKD_SHA3_384_KDF
CKD_SHA3_512_KDF
CKD_SHA1_KDF_SP800
CKD_SHA224_KDF_SP800
CKD_SHA256_KDF_SP800
CKD_SHA384_KDF_SP800
CKD_SHA512_KDF_SP800
CKD_SHA3_224_KDF_SP800
CKD_SHA3_256_KDF_SP800
CKD_SHA3_384_KDF_SP800
CKD_SHA3_512_KDF_SP800
CKD_BLAKE2B_160_KDF
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_384_KDF
CKD_BLAKE2B_512_KDF

1744 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
 1745 derivation function.

1746 The key derivation functions
 1747 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**, which are
 1748 based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
 1749 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

1750 The key derivation functions
 1751 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**,
 1752 which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
 1753 respectively, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section
 1754 5.8.1.1.

1755 The key derivation functions **CKD_BLAKE2B_[160|256|384|512]_KDF**, which are based on the Blake2b
 1756 family of hashes, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section
 1757 5.8.1.1.**CK_EC_KDF_TYPE_PTR** is a pointer to a **CK_EC_KDF_TYPE**.

1758

1759 ◆ **CK_ECDH1_DERIVE_PARAMS, CK_ECDH1_DERIVE_PARAMS_PTR**

1760 **CK_ECDH1_DERIVE_PARAMS** is a structure that provides the parameters for the
 1761 **CKM_ECDH1_DERIVE** and **CKM_ECDH1_COFACTOR_DERIVE** key derivation mechanisms, where
 1762 each party contributes one key pair. The structure is defined as follows:

```
1763     typedef struct CK_ECDH1_DERIVE_PARAMS {
1764         CK_EC_KDF_TYPE    kdf;
1765         CK ULONG          ulSharedDataLen;
1766         CK BYTE PTR       pSharedData;
1767         CK ULONG          ulPublicDataLen;
1768         CK BYTE PTR       pPublicData;
1769     } CK_ECDH1_DERIVE_PARAMS;
```

1770

1771 The fields of the structure have the following meanings:

1772 *kdf* *key derivation function used on the shared secret value*

1773	<i>ulSharedDataLen</i>	<i>the length in bytes of the shared info</i>
1774	<i>pSharedData</i>	<i>some data shared between the two parties</i>
1775	<i>ulPublicDataLen</i>	<i>the length in bytes of the other party's EC public key</i>
1776	<i>pPublicData</i> ¹	<i>pointer to other party's EC public key value. A token MUST be able to accept this value encoded as a raw octet string (as per section A.5.2 of [ANSI X9.62]). A token MAY, in addition, support accepting this value as a DER-encoded ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a CKA_EC_POINT encoding. The calling application is responsible for converting the offered public key to the compressed or uncompressed forms of these encodings if the token does not support the offered form.</i>
1784	With the key derivation function CKD_NULL , <i>pSharedData</i> must be NULL and <i>ulSharedDataLen</i> must be zero. With the key derivation functions	
1786	CKD_[SHA1 SHA224 SHA384 SHA512 SHA3_224 SHA3_256 SHA3_384 SHA3_512]_KDF ,	
1787	CKD_[SHA1 SHA224 SHA384 SHA512 SHA3_224 SHA3_256 SHA3_384 SHA3_512]_KDF_SP800 , an	
1788	optional <i>pSharedData</i> may be supplied, which consists of some data shared by the two parties intending	
1789	to share the shared secret. Otherwise, <i>pSharedData</i> must be NULL and <i>ulSharedDataLen</i> must be zero.	
1790	CK_ECDH1_DERIVE_PARAMS_PTR is a pointer to a CK_ECDH1_DERIVE_PARAMS .	
1791	◆ CK_ECDH2_DERIVE_PARAMS , CK_ECDH2_DERIVE_PARAMS_PTR	
1792	CK_ECDH2_DERIVE_PARAMS is a structure that provides the parameters to the	
1793	CKM_ECMQV_DERIVE key derivation mechanism, where each party contributes two key pairs. The	
1794	structure is defined as follows:	
1795	typedef struct CK_ECDH2_DERIVE_PARAMS {	
1796	CK_EC_KDF_TYPE kdf;	
1797	CK ULONG ulSharedDataLen;	
1798	CK_BYTE_PTR pSharedData;	
1799	CK ULONG ulPublicDataLen;	
1800	CK_BYTE_PTR pPublicData;	
1801	CK ULONG ulPrivateDataLen;	
1802	CK_OBJECT_HANDLE hPrivateKey;	
1803	CK ULONG ulPublicDataLen2;	
1804	CK_BYTE_PTR pPublicData2;	
1805	}	
1806	CK_ECDH2_DERIVE_PARAMS ;	
1807	The fields of the structure have the following meanings:	
1808	<i>kdf</i>	<i>key derivation function used on the shared secret value</i>
1809	<i>ulSharedDataLen</i>	<i>the length in bytes of the shared info</i>
1810	<i>pSharedData</i>	<i>some data shared between the two parties</i>
1811	<i>ulPublicDataLen</i>	<i>the length in bytes of the other party's first EC public key</i>

1. The encoding in V2.20 was not specified and resulted in different implementations choosing different encodings. Applications relying only on a V2.20 encoding (e.g. the DER variant) other than the one specified now (raw) may not work with all V2.30 compliant tokens.

1812 *pPublicData* pointer to other party's first EC public key value. Encoding rules are
 1813 as per *pPublicData* of CK_ECDH1_DERIVE_PARAMS
 1814 *ulPrivateDataLen* the length in bytes of the second EC private key
 1815 *hPrivateKeyData* key handle for second EC private key
 1816 *ulPublicDataLen2* the length in bytes of the other party's second EC public key
 1817 *pPublicData2* pointer to other party's second EC public key value. Encoding rules
 1818 are as per *pPublicData* of CK_ECDH1_DERIVE_PARAMS

1819 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be
 1820 zero. With the key derivation function **CKD_SHA1_KDF**, an optional *pSharedData* may be supplied,
 1821 which consists of some data shared by the two parties intending to share the shared secret. Otherwise,
 1822 *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

1823 **CK_ECDH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECDH2_DERIVE_PARAMS**.
 1824

1825 ◆ **CK_ECMQV_DERIVE_PARAMS, CK_ECMQV_DERIVE_PARAMS_PTR**

1826 **CK_ECMQV_DERIVE_PARAMS** is a structure that provides the parameters to the
 1827 **CKM_ECMQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
 1828 structure is defined as follows:

```

 1829     typedef struct CK_ECMQV_DERIVE_PARAMS {
 1830         CK_EC_KDF_TYPE      kdf;
 1831         CK ULONG            ulSharedDataLen;
 1832         CK_BYTE_PTR         pSharedData;
 1833         CK ULONG            ulPublicDataLen;
 1834         CK_BYTE_PTR         pPublicData;
 1835         CK ULONG            ulPrivateDataLen;
 1836         CK_OBJECT_HANDLE    hPrivateKeyData;
 1837         CK ULONG            ulPublicDataLen2;
 1838         CK_BYTE_PTR         pPublicData2;
 1839         CK_OBJECT_HANDLE    publicKey;
 1840     } CK_ECMQV_DERIVE_PARAMS;
 1841
  
```

1842 The fields of the structure have the following meanings:

1843 <i>kdf</i> 1844 <i>ulSharedDataLen</i> 1845 <i>pSharedData</i> 1846 <i>ulPublicDataLen</i> 1847 <i>pPublicData</i> 1848 <i>ulPrivateDataLen</i> 1849 <i>hPrivateKeyData</i>	<i>key derivation function used on the shared secret value</i> <i>the length in bytes of the shared info</i> <i>some data shared between the two parties</i> <i>the length in bytes of the other party's first EC public key</i> <i>pointer to other party's first EC public key value. Encoding rules are as per <i>pPublicData</i> of CK_ECDH1_DERIVE_PARAMS</i> <i>the length in bytes of the second EC private key</i> <i>key handle for second EC private key value</i>
--	--

1851	<i>ulPublicDataLen2</i>	<i>the length in bytes of the other party's second EC public key</i>
1852	<i>pPublicData2</i>	<i>pointer to other party's second EC public key value. Encoding rules are as per pPublicData of CK_ECDH1_DERIVE_PARAMS</i>
1853		
1854	<i>publicKey</i>	<i>Handle to the first party's ephemeral public key</i>
1855	With the key derivation function CKD_NULL , <i>pSharedData</i> must be NULL and <i>ulSharedDataLen</i> must be zero. With the key derivation functions	
1856	CKD_[SHA1 SHA224 SHA384 SHA512 SHA3_224 SHA3_256 SHA3_384 SHA3_512]_KDF ,	
1857	CKD_[SHA1 SHA224 SHA384 SHA512 SHA3_224 SHA3_256 SHA3_384 SHA3_512]_KDF_SP800 , an	
1858	optional <i>pSharedData</i> may be supplied, which consists of some data shared by the two parties intending	
1859	to share the shared secret. Otherwise, <i>pSharedData</i> must be NULL and <i>ulSharedDataLen</i> must be zero.	
1860		
1861	CK_ECMQV_DERIVE_PARAMS_PTR is a pointer to a CK_ECMQV_DERIVE_PARAMS .	

2.3.17 Elliptic curve Diffie-Hellman key derivation

1862	The elliptic curve Diffie-Hellman (ECDH) key derivation mechanism, denoted CKM_ECDH1_DERIVE , is a mechanism for key derivation based on the Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters.
1863	It has a parameter, a CK_ECDH1_DERIVE_PARAMS structure.
1864	This mechanism derives a secret value, and truncates the result according to the CKA_KEY_TYPE attribute of the template and, if it has one and the key type supports it, the CKA_VALUE_LEN attribute of
1865	the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
1866	contributes the result as the CKA_VALUE attribute of the new key; other attributes required by the key
1867	type must be specified in the template.
1868	This mechanism has the following rules about key sensitivity and extractability:
1869	<ul style="list-style-type: none"> The CKA_SENSITIVE and CKA_EXTRACTABLE attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.
1870	<ul style="list-style-type: none"> If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_FALSE, then the derived key will as well. If the base key has its CKA_ALWAYS_SENSITIVE attribute set to CK_TRUE, then the derived key has its CKA_ALWAYS_SENSITIVE attribute set to the same value as its CKA_SENSITIVE attribute.
1871	<ul style="list-style-type: none"> Similarly, if the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_FALSE, then the derived key will, too. If the base key has its CKA_NEVER_EXTRACTABLE attribute set to CK_TRUE, then the derived key has its CKA_NEVER_EXTRACTABLE attribute set to the <i>opposite</i> value from its CKA_EXTRACTABLE attribute.
1872	For this mechanism, the <i>ulMinKeySize</i> and <i>ulMaxKeySize</i> fields of the CK_MECHANISM_INFO structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For
1873	example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2^{200} and 2^{300} elements, then <i>ulMinKeySize</i> = 201 and <i>ulMaxKeySize</i> = 301 (when written in binary notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2^{300} is a 301-bit number).
1874	Constraints on key types are summarized in the following table:
1875	
1876	
1877	
1878	
1879	
1880	
1881	
1882	
1883	
1884	
1885	
1886	
1887	
1888	
1889	
1890	
1891	

1892 *Table 48: ECDH: Allowed Key Types*

Function	Key type
C_Derive	CKK_EC or CKK_EC_MONTGOMERY

2.3.18 Elliptic curve Diffie-Hellman with cofactor key derivation

The elliptic curve Diffie-Hellman (ECDH) with cofactor key derivation mechanism, denoted **CKM_ECDH1_COFACTOR_DERIVE**, is a mechanism for key derivation based on the cofactor Diffie-Hellman version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes one key pair all using the same EC domain parameters. Cofactor multiplication is computationally efficient and helps to prevent security problems like small group attacks.

It has a parameter, a **CK_ECDH1_DERIVE_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key type must be specified in the template.

This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its **CKA_EXTRACTABLE** attribute.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the minimum and maximum supported number of bits in the field sizes, respectively. For example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2^{200} and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2^{300} is a 301-bit number).

Constraints on key types are summarized in the following table:

1924 *Table 49: ECDH with cofactor: Allowed Key Types*

Function	Key type
C_Derive	CKK_EC

2.3.19 Elliptic curve Menezes-Qu-Vanstone key derivation

The elliptic curve Menezes-Qu-Vanstone (ECMQV) key derivation mechanism, denoted **CKM_ECMQV_DERIVE**, is a mechanism for key derivation based the MQV version of the elliptic curve key agreement scheme, as defined in ANSI X9.63, where each party contributes two key pairs all using the same EC domain parameters.

It has a parameter, a **CK_ECMQV_DERIVE_PARAMS** structure.

This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism

1934 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
 1935 type must be specified in the template.
 1936 This mechanism has the following rules about key sensitivity and extractability:
 1937 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
 1938 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
 1939 default value.
 1940 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
 1941 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
 1942 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 1943 **CKA_SENSITIVE** attribute.
 1944 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
 1945 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 1946 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 1947 value from its **CKA_EXTRACTABLE** attribute.
 1948 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 1949 specify the minimum and maximum supported number of bits in the field sizes, respectively. For
 1950 example, if a Cryptoki library supports only EC using a field of characteristic 2 which has between 2^{200}
 1951 and 2^{300} elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation,
 1952 the number 2^{200} consists of a 1 bit followed by 200 0 bits. It is therefore a 201-bit number. Similarly, 2^{300}
 1953 is a 301-bit number).
 1954 Constraints on key types are summarized in the following table:
 1955 *Table 50: ECDH MQV: Allowed Key Types*

Function	Key type
C_Derive	CKK_EC

1956 **2.3.20 ECDH AES KEY WRAP**
 1957 The ECDH AES KEY WRAP mechanism, denoted **CKM_ECDH_AES_KEY_WRAP**, is a mechanism
 1958 based on elliptic curve public-key crypto-system and the AES key wrap mechanism. It supports single-
 1959 part key wrapping; and key unwrapping.
 1960 It has a parameter, a **CK_ECDH_AES_KEY_WRAP_PARAMS** structure.
 1961
 1962 The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC
 1963 key.
 1964 - A temporary AES key is derived from a temporary EC key and the wrapping EC key
 1965 using the **CKM_ECDH1_DERIVE** mechanism.
 1966 - The derived AES key is used for wrapping the target key using the
 1967 **CKM_AES_KEY_WRAP_KWP** mechanism.
 1968
 1969 For wrapping, the mechanism -
 1970 • Generates a temporary random EC key (transport key) having the same parameters as the
 1971 wrapping EC key (and domain parameters). Saves the transport key public key material.
 1972 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf,
 1973
 1974 key of wrapping EC key and gets the first ulAESKeyBits bits of the derived key to be the
 1975 temporary AES key.
 1976 • Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP** ([AES
 1977 KEYWRAP] section 6.3).
 1978 • Zeroizes the temporary AES key and EC transport private key.

- 1979 • Concatenates public key material of the transport key and output the concatenated blob. The first
 1980 part is the public key material of the transport key and the second part is the wrapped target key.

1981
 1982 The recommended format for an asymmetric target key being wrapped is as a PKCS8
 1983 PrivateKeyInfo

1984
 1985 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the
 1986 object attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.

1987
 1988 For unwrapping, the mechanism -

- 1989 • Splits the input into two parts. The first part is the public key material of the transport key and the
 1990 second part is the wrapped target key. The length of the first part is equal to the length of the
 1991 public key material of the unwrapping EC key.

1992 *Note: since the transport key and the wrapping EC key share the same domain, the length of the
 1993 public key material of the transport key is the same length of the public key material of the
 1994 unwrapping EC key.*

- 1995 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf,
 1996 ulSharedDataLen and pSharedData using the private part of unwrapping EC key and the public
 1997 part of the transport EC key and gets first ulAESKeyBits bits of the derived key to be the
 1998 temporary AES key.
- 1999 • Un-wraps the target key from the second part with the temporary AES key using
 2000 **CKM_AES_KEY_WRAP_KWP** ([AES KEYWRAP] section 6.3).
- 2001 • Zeroizes the temporary AES key.

2002
 2003 Table 51, CKM_ECDH_AES_KEY_WRAP Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ECDH_AES_KEY_WRAP						✓	

¹SR = SignRecover, VR = VerifyRecover

2004
 2005 Constraints on key types are summarized in the following table:

2006 Table 52: ECDH AES Key Wrap: Allowed Key Types

Function	Key type
C_Derive	CKK_EC or CKK_EC_MONTGOMERY

2007 2.3.21 ECDH AES KEY WRAP mechanism parameters

- 2008 ♦ **CK_ECDH_AES_KEY_WRAP_PARAMS; CK_ECDH_AES_KEY_WRAP_PARAMS_PTR**

2009 **CK_ECDH_AES_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
 2010 **CKM_ECDH_AES_KEY_WRAP** mechanism. It is defined as follows:

2011
 2012 typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
 2013 CK ULONG ulAESKeyBits;

```

2014     CK_EC_KDF_TYPE    kdf;
2015     CK ULONG          ulSharedDataLen;
2016     CK_BYTE_PTR       pSharedData;
2017 } CK_ECDH_AES_KEY_WRAP_PARAMS;
2018
2019 The fields of the structure have the following meanings:
2020
2021     ulAESKeyBits      length of the temporary AES key in bits. Can be only 128, 192 or
2022                     256.
2023
2024     kdf               key derivation function used on the shared secret value to generate
2025                     AES key.
2026
2027     ulSharedDataLen    the length in bytes of the shared info
2028
2029     pSharedData        Some data shared between the two parties
2030

```

2031 **2.3.22 FIPS 186-4**

2032 When CKM_ECDSA is operated in FIPS mode, the curves SHALL either be NIST recommended curves
 2033 (with a fixed set of domain parameters) or curves with domain parameters generated as specified by
 2034 ANSI X9.64. The NIST recommended curves are:

2035
 2036 P-192, P-224, P-256, P-384, P-521
 2037 K-163, B-163, K-233, B-233
 2038 K-283, B-283, K-409, B-409
 2039 K-571, B-571

2040 **2.4 Diffie-Hellman**

2041 *Table 53, Diffie-Hellman Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen . Key / Key Pair	Wrap & Unwrap	Deriv e
CKM_DH_PKCS_KEY_PAIR_GEN					✓		
CKM_DH_PKCS_PARAMETER_GEN					✓		
CKM_DH_PKCS_DERIVE							✓
CKM_X9_42_DH_KEY_PAIR_GEN					✓		
CKM_X9_42_DH_PKCS_PARAMETER_GEN					✓		

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify ¹	SR & VR ¹	Digest	Gen. Key / Key Pair	Wrap & Unwrap	Derive
CKM_X9_42_DH_DERIVE							✓
CKM_X9_42_DH_HYBRID_DERIVE							✓
CKM_X9_42_MQV_DERIVE							✓

2.4.1 Definitions

This section defines the key type “CKK_DH” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of [DH] key objects.

Mechanisms:

- CKM_DH_PKCS_KEY_PAIR_GEN
- CKM_DH_PKCS_DERIVE
- CKM_X9_42_DH_KEY_PAIR_GEN
- CKM_X9_42_DH_DERIVE
- CKM_X9_42_DH_HYBRID_DERIVE
- CKM_X9_42_MQV_DERIVE
- CKM_DH_PKCS_PARAMETER_GEN
- CKM_X9_42_DH_PARAMETER_GEN

2.4.2 Diffie-Hellman public key objects

Diffie-Hellman public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_DH**) hold Diffie-Hellman public keys. The following table defines the Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table 54, Diffie-Hellman Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME ^{1,3}	Big integer	Prime p
CKA_BASE ^{1,3}	Big integer	Base g
CKA_VALUE ^{1,4}	Big integer	Public value y

- Refer to [PKCS11-Base] table 11 for footnotes

The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain parameters”. Depending on the token, there may be limits on the length of the key components. See PKCS #3 for more information on Diffie-Hellman keys.

The following is a sample template for creating a Diffie-Hellman public key object:

```
2064 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
2065 CK_KEY_TYPE keyType = CKK_DH;
2066 CK_UTF8CHAR label[] = "A Diffie-Hellman public key object";
2067 CK_BYTE prime[] = { ... };
2068 CK_BYTE base[] = { ... };
2069 CK_BYTE value[] = { ... };
```

```

2070    CK_BBOOL true = CK_TRUE;
2071    CK_ATTRIBUTE template[] = {
2072        {CKA_CLASS, &class, sizeof(class)},
2073        {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2074        {CKA_TOKEN, &true, sizeof(true)},
2075        {CKA_LABEL, label, sizeof(label)-1},
2076        {CKA_PRIME, prime, sizeof(prime)},
2077        {CKA_BASE, base, sizeof(base)},
2078        {CKA_VALUE, value, sizeof(value)}}
2079    };
```

2.4.3 X9.42 Diffie-Hellman public key objects

X9.42 Diffie-Hellman public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_X9_42_DH**) hold X9.42 Diffie-Hellman public keys. The following table defines the X9.42 Diffie-Hellman public key object attributes, in addition to the common attributes defined for this object class:

Table 55, X9.42 Diffie-Hellman Public Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME ^{1,3}	Big integer	Prime p (≥ 1024 bits, in steps of 256 bits)
CKA_BASE ^{1,3}	Big integer	Base g
CKA_SUBPRIME ^{1,3}	Big integer	Subprime q (≥ 160 bits)
CKA_VALUE ^{1,4}	Big integer	Public value y

2085 - Refer to [PKCS11-Base] table 11 for footnotes

2086 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
2087 Hellman domain parameters”. See the ANSI X9.42 standard for more information on X9.42 Diffie-
2088 Hellman keys.

2089 The following is a sample template for creating a X9.42 Diffie-Hellman public key object:

```

2090    CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
2091    CK_KEY_TYPE keyType = CKK_X9_42_DH;
2092    CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key
2093        object";
2094    CK_BYTE prime[] = {...};
2095    CK_BYTE base[] = {...};
2096    CK_BYTE subprime[] = {...};
2097    CK_BYTE value[] = {...};
2098    CK_BBOOL true = CK_TRUE;
2099    CK_ATTRIBUTE template[] = {
2100        {CKA_CLASS, &class, sizeof(class)},
2101        {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2102        {CKA_TOKEN, &true, sizeof(true)},
2103        {CKA_LABEL, label, sizeof(label)-1},
2104        {CKA_PRIME, prime, sizeof(prime)},
2105        {CKA_BASE, base, sizeof(base)},
2106        {CKA_SUBPRIME, subprime, sizeof(subprime)},
2107        {CKA_VALUE, value, sizeof(value)}}
2108    };
```

2109 2.4.4 Diffie-Hellman private key objects

2110 Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_DH**) hold Diffie-
2111 Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in
2112 addition to the common attributes defined for this object class:

2113 *Table 56, Diffie-Hellman Private Key Object Attributes*

Attribute	Data type	Meaning
CKA_PRIME ^{1,4,6}	Big integer	Prime p
CKA_BASE ^{1,4,6}	Big integer	Base g
CKA_VALUE ^{1,4,6,7}	Big integer	Private value x
CKA_VALUE_BITS ^{2,6}	CK ULONG	Length in bits of private value x

2114 - Refer to [PKCS11-Base] table 11 for footnotes

2115 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
2116 parameters”. Depending on the token, there may be limits on the length of the key components. See
2117 PKCS #3 for more information on Diffie-Hellman keys.

2118 Note that when generating a Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in
2119 the key’s template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-
2120 Hellman key pair, and the Diffie-Hellman parameters for the pair are specified in the template for the
2121 Diffie-Hellman public key.

2122 The following is a sample template for creating a Diffie-Hellman private key object:

```
2123 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
2124 CK_KEY_TYPE keyType = CKK_DH;
2125 CK_UTF8CHAR label[] = "A Diffie-Hellman private key object";
2126 CK_BYTE subject[] = {...};
2127 CK_BYTE id[] = {123};
2128 CK_BYTE prime[] = {...};
2129 CK_BYTE base[] = {...};
2130 CK_BYTE value[] = {...};
2131 CK_BBOOL true = CK_TRUE;
2132 CK_ATTRIBUTE template[] = {
2133     {CKA_CLASS, &class, sizeof(class)},
2134     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2135     {CKA_TOKEN, &true, sizeof(true)},
2136     {CKA_LABEL, label, sizeof(label)-1},
2137     {CKA_SUBJECT, subject, sizeof(subject)},
2138     {CKA_ID, id, sizeof(id)},
2139     {CKA_SENSITIVE, &true, sizeof(true)},
2140     {CKA_DERIVE, &true, sizeof(true)},
2141     {CKA_PRIME, prime, sizeof(prime)},
2142     {CKA_BASE, base, sizeof(base)},
2143     {CKA_VALUE, value, sizeof(value)}
2144 };
```

2145 2.4.5 X9.42 Diffie-Hellman private key objects

2146 X9.42 Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_X9_42_DH**)
2147 hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key
2148 object attributes, in addition to the common attributes defined for this object class:

2149 Table 57, X9.42 Diffie-Hellman Private Key Object Attributes

Attribute	Data type	Meaning
CKA_PRIME ^{1,4,6}	Big integer	Prime p (≥ 1024 bits, in steps of 256 bits)
CKA_BASE ^{1,4,6}	Big integer	Base g
CKA_SUBPRIME ^{1,4,6}	Big integer	Subprime q (≥ 160 bits)
CKA_VALUE ^{1,4,6,7}	Big integer	Private value x

2150 - Refer to [PKCS11-Base] table 11 for footnotes

2151 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
2152 Hellman domain parameters”. Depending on the token, there may be limits on the length of the key
2153 components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.2154 Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain
2155 parameters are *not* specified in the key's template. This is because X9.42 Diffie-Hellman private keys are
2156 only generated as part of a X9.42 Diffie-Hellman key pair, and the X9.42 Diffie-Hellman domain
2157 parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

2158 The following is a sample template for creating a X9.42 Diffie-Hellman private key object:

```

2159     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
2160     CK_KEY_TYPE keyType = CKK_X9_42_DH;
2161     CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman private key object";
2162     CK_BYTE subject[] = {...};
2163     CK_BYTE id[] = {123};
2164     CK_BYTE prime[] = {...};
2165     CK_BYTE base[] = {...};
2166     CK_BYTE subprime[] = {...};
2167     CK_BYTE value[] = {...};
2168     CK_BBOOL true = CK_TRUE;
2169     CK_ATTRIBUTE template[] = {
2170         {CKA_CLASS, &class, sizeof(class)},
2171         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2172         {CKA_TOKEN, &true, sizeof(true)},
2173         {CKA_LABEL, label, sizeof(label)-1},
2174         {CKA_SUBJECT, subject, sizeof(subject)},
2175         {CKA_ID, id, sizeof(id)},
2176         {CKA_SENSITIVE, &true, sizeof(true)},
2177         {CKA_DERIVE, &true, sizeof(true)},
2178         {CKA_PRIME, prime, sizeof(prime)},
2179         {CKA_BASE, base, sizeof(base)},
2180         {CKA_SUBPRIME, subprime, sizeof(subprime)},
2181         {CKA_VALUE, value, sizeof(value)}
2182     };

```

2183

2.4.6 Diffie-Hellman domain parameter objects

2184 Diffie-Hellman domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
2185 **CKK_DH**) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain
2186 parameter object attributes, in addition to the common attributes defined for this object class:

2187 Table 58, Diffie-Hellman Domain Parameter Object Attributes

Attribute	Data type	Meaning
CKA_PRIME ^{1,4}	Big integer	Prime p
CKA_BASE ^{1,4}	Big integer	Base g
CKA_PRIME_BITS ^{2,3}	CK ULONG	Length of the prime value.

2188 - Refer to [PKCS11-Base] table 11 for footnotes

2189 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
 2190 parameters”. Depending on the token, there may be limits on the length of the key components. See
 2191 PKCS #3 for more information on Diffie-Hellman domain parameters.

2192 The following is a sample template for creating a Diffie-Hellman domain parameter object:

```

2193 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
2194 CK_KEY_TYPE keyType = CKK_DH;
2195 CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters
2196         object";
2197 CK_BYTE prime[] = {...};
2198 CK_BYTE base[] = {...};
2199 CK_BBOOL true = CK_TRUE;
2200 CK_ATTRIBUTE template[] = {
2201     {CKA_CLASS, &class, sizeof(class)},
2202     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2203     {CKA_TOKEN, &true, sizeof(true)},
2204     {CKA_LABEL, label, sizeof(label)-1},
2205     {CKA_PRIME, prime, sizeof(prime)},
2206     {CKA_BASE, base, sizeof(base)}},
2207 };
  
```

2208 2.4.7 X9.42 Diffie-Hellman domain parameters objects

2209 X9.42 Diffie-Hellman domain parameters objects (object class **CKO_DOMAIN_PARAMETERS**, key type
 2210 **CKK_X9_42_DH**) hold X9.42 Diffie-Hellman domain parameters. The following table defines the X9.42
 2211 Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this
 2212 object class:

2213 Table 59, X9.42 Diffie-Hellman Domain Parameters Object Attributes

Attribute	Data type	Meaning
CKA_PRIME ^{1,4}	Big integer	Prime p (≥ 1024 bits, in steps of 256 bits)
CKA_BASE ^{1,4}	Big integer	Base g
CKA_SUBPRIME ^{1,4}	Big integer	Subprime q (≥ 160 bits)
CKA_PRIME_BITS ^{2,3}	CK ULONG	Length of the prime value.
CKA_SUBPRIME_BITS ^{2,3}	CK ULONG	Length of the subprime value.

2214 - Refer to [PKCS11-Base] table 11 for footnotes

2215 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
 2216 Hellman domain parameters”. Depending on the token, there may be limits on the length of the domain
 2217 parameters components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman
 2218 domain parameters.

2219 The following is a sample template for creating a X9.42 Diffie-Hellman domain parameters object:

```

2220 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
2221 CK_KEY_TYPE keyType = CKK_X9_42_DH;
  
```

```

2222     CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain
2223         parameters object";
2224     CK_BYTE prime[] = {...};
2225     CK_BYTE base[] = {...};
2226     CK_BYTE subprime[] = {...};
2227     CK_BBOOL true = CK_TRUE;
2228     CK_ATTRIBUTE template[] = {
2229         {CKA_CLASS, &class, sizeof(class)},
2230         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2231         {CKA_TOKEN, &true, sizeof(true)},
2232         {CKA_LABEL, label, sizeof(label)-1},
2233         {CKA_PRIME, prime, sizeof(prime)},
2234         {CKA_BASE, base, sizeof(base)},
2235         {CKA_SUBPRIME, subprime, sizeof(subprime)},
2236     };

```

2237 2.4.8 PKCS #3 Diffie-Hellman key pair generation

2238 The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted
2239 **CKM_DH_PKCS_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key
2240 agreement, as defined in PKCS #3. This is what PKCS #3 calls “phase I”. It does not have a parameter.
2241 The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as
2242 specified in the **CKA_PRIME** and **CKA_BASE** attributes of the template for the public key. If the
2243 **CKA_VALUE_BITS** attribute of the private key is specified, the mechanism limits the length in bits of the
2244 private value, as described in PKCS #3.
2245 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
2246 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and **CKA_VALUE** (and
2247 the **CKA_VALUE_BITS** attribute, if it is not already provided in the template) attributes to the new private
2248 key; other attributes required by the Diffie-Hellman public and private key types must be specified in the
2249 templates.
2250 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2251 specify the supported range of Diffie-Hellman prime sizes, in bits.

2252 2.4.9 PKCS #3 Diffie-Hellman domain parameter generation

2253 The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted
2254 **CKM_DH_PKCS_PARAMETER_GEN**, is a domain parameter generation mechanism based on Diffie-
2255 Hellman key agreement, as defined in PKCS #3.
2256 It does not have a parameter.
2257 The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as
2258 specified in the **CKA_PRIME_BITS** attribute of the template.
2259 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and
2260 **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the Diffie-Hellman domain
2261 parameter types may also be specified in the template, or else are assigned default initial values.
2262 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2263 specify the supported range of Diffie-Hellman prime sizes, in bits.

2264 2.4.10 PKCS #3 Diffie-Hellman key derivation

2265 The PKCS #3 Diffie-Hellman key derivation mechanism, denoted **CKM_DH_PKCS_DERIVE**, is a
2266 mechanism for key derivation based on Diffie-Hellman key agreement, as defined in PKCS #3. This is
2267 what PKCS #3 calls “phase II”.

2268 It has a parameter, which is the public value of the other party in the key agreement protocol, represented
2269 as a Cryptoki "Big integer" (i.e., a sequence of bytes, most-significant byte first).

2270 This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other
2271 party. It computes a Diffie-Hellman secret value from the public value and private key according to PKCS
2272 #3, and truncates the result according to the **CKA_KEY_TYPE** attribute of the template and, if it has one
2273 and the key type supports it, the **CKA_VALUE_LEN** attribute of the template. (The truncation removes
2274 bytes from the leading end of the secret value.) The mechanism contributes the result as the
2275 **CKA_VALUE** attribute of the new key; other attributes required by the key type must be specified in the
2276 template.

2277 This mechanism has the following rules about key sensitivity and extractability²:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
2279 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
2280 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
2281 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
2282 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
2283 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
2285 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
2286 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
2287 value from its **CKA_EXTRACTABLE** attribute.

2289 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2290 specify the supported range of Diffie-Hellman prime sizes, in bits.

2291 2.4.11 X9.42 Diffie-Hellman mechanism parameters

2292 ♦ **CK_X9_42_DH_KDF_TYPE**, **CK_X9_42_DH_KDF_TYPE_PTR**

2293 **CK_X9_42_DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive
2294 keying data from a shared secret. The key derivation function will be used by the X9.42 Diffie-Hellman
2295 key agreement schemes. It is defined as follows:

2296 `typedef CK ULONG CK_X9_42_DH_KDF_TYPE;`

2298 The following table lists the defined functions.

2299 *Table 60, X9.42 Diffie-Hellman Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_SHA1_KDF ASN1
CKD_SHA1_KDF CONCATENATE

2300 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
2301 derivation function whereas the key derivation functions **CKD_SHA1_KDF ASN1** and
2302 **CKD_SHA1_KDF CONCATENATE**, which are both based on SHA-1, derive keying data from the
2303 shared secret value as defined in the ANSI X9.42 standard.

2304 **CK_X9_42_DH_KDF_TYPE_PTR** is a pointer to a **CK_X9_42_DH_KDF_TYPE**.

2 Note that the rules regarding the CKA_SENSITIVE, CKA_EXTRACTABLE, CKA_ALWAYS_SENSITIVE, and CKA_NEVER_EXTRACTABLE attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as CKM_SSL3_MASTER_KEY_DERIVE.

2305 ◆ **CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR**

2306 **CK_X9_42_DH1_DERIVE_PARAMS** is a structure that provides the parameters to the
2307 **CKM_X9_42_DH_DERIVE** key derivation mechanism, where each party contributes one key pair. The
2308 structure is defined as follows:

```
2309     typedef struct CK_X9_42_DH1_DERIVE_PARAMS {  
2310         CK_X9_42_DH_KDF_TYPE    kdf;  
2311         CK ULONG                ulOtherInfoLen;  
2312         CK BYTE PTR             pOtherInfo;  
2313         CK ULONG                ulPublicDataLen;  
2314         CK BYTE PTR             pPublicData;  
2315     } CK_X9_42_DH1_DERIVE_PARAMS;
```

2316 The fields of the structure have the following meanings:

2318 *kdf* key derivation function used on the shared secret value

2319 *ulOtherInfoLen* the length in bytes of the other info

2320 *pOtherInfo* some data shared between the two parties

2321 *ulPublicDataLen* the length in bytes of the other party's X9.42 Diffie-Hellman public
2322 key

2323 *pPublicData* pointer to other party's X9.42 Diffie-Hellman public key value

2324 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
2325 With the key derivation function **CKD_SHA1_KDF ASN1**, *pOtherInfo* must be supplied, which contains
2326 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
2327 the two parties intending to share the shared secret. With the key derivation function
2328 **CKD_SHA1_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
2329 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
2330 NULL and *ulOtherInfoLen* must be zero.

2331 **CK_X9_42_DH1_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH1_DERIVE_PARAMS**.

2332 • **CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR**

2333 **CK_X9_42_DH2_DERIVE_PARAMS** is a structure that provides the parameters to the
2334 **CKM_X9_42_DH_HYBRID_DERIVE** and **CKM_X9_42_MQV_DERIVE** key derivation mechanisms,
2335 where each party contributes two key pairs. The structure is defined as follows:

```
2336     typedef struct CK_X9_42_DH2_DERIVE_PARAMS {  
2337         CK_X9_42_DH_KDF_TYPE    kdf;  
2338         CK ULONG                ulOtherInfoLen;  
2339         CK BYTE PTR             pOtherInfo;  
2340         CK ULONG                ulPublicDataLen;  
2341         CK BYTE PTR             pPublicData;  
2342         CK ULONG                ulPrivateDataLen;  
2343         CK OBJECT_HANDLE       hPrivateData;  
2344         CK ULONG                ulPublicDataLen2;  
2345         CK BYTE PTR             pPublicData2;  
2346     } CK_X9_42_DH2_DERIVE_PARAMS;
```

2347

2348 The fields of the structure have the following meanings:

2349	<i>kdf</i>	<i>key derivation function used on the shared secret value</i>
2350	<i>ulOtherInfoLen</i>	<i>the length in bytes of the other info</i>
2351	<i>pOtherInfo</i>	<i>some data shared between the two parties</i>
2352	<i>ulPublicDataLen</i>	<i>the length in bytes of the other party's first X9.42 Diffie-Hellman public key</i>
2353		
2354	<i>pPublicData</i>	<i>pointer to other party's first X9.42 Diffie-Hellman public key value</i>
2355	<i>ulPrivateDataLen</i>	<i>the length in bytes of the second X9.42 Diffie-Hellman private key</i>
2356	<i>hPrivateKey</i>	<i>key handle for second X9.42 Diffie-Hellman private key value</i>
2357	<i>ulPublicDataLen2</i>	<i>the length in bytes of the other party's second X9.42 Diffie-Hellman public key</i>
2358		
2359	<i>pPublicData2</i>	<i>pointer to other party's second X9.42 Diffie-Hellman public key value</i>
2360		

2361 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.

2362 With the key derivation function **CKD_SHA1_KDF ASN1**, *pOtherInfo* must be supplied, which contains

2363 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by

2364 the two parties intending to share the shared secret. With the key derivation function

2365 **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some

2366 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be

2367 NULL and *ulOtherInfoLen* must be zero.

2368 **CK_X9_42_DH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH2_DERIVE_PARAMS**.

2369 • **CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR**

2370 **CK_X9_42_MQV_DERIVE_PARAMS** is a structure that provides the parameters to the

2371 **CKM_X9_42_MQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The

2372 structure is defined as follows:

```

2373     typedef struct CK_X9_42_MQV_DERIVE_PARAMS {
2374         CK_X9_42_DH_KDF_TYPE kdf;
2375         CK ULONG ulOtherInfoLen;
2376         CK BYTE PTR pOtherInfo;
2377         CK ULONG ulPublicDataLen;
2378         CK BYTE PTR pPublicData;
2379         CK ULONG ulPrivateDataLen;
2380         CK OBJECT_HANDLE hPrivateKey;
2381         CK ULONG ulPublicDataLen2;
2382         CK BYTE PTR pPublicData2;
2383         CK OBJECT_HANDLE publicKey;
2384     } CK_X9_42_MQV_DERIVE_PARAMS;

```

2385
2386 The fields of the structure have the following meanings:
2387 *kdf* *key derivation function used on the shared secret value*
2388 *ulOtherInfoLen* *the length in bytes of the other info*
2389 *pOtherInfo* *some data shared between the two parties*
2390 *ulPublicDataLen* *the length in bytes of the other party's first X9.42 Diffie-Hellman public key*
2391
2392 *pPublicData* *pointer to other party's first X9.42 Diffie-Hellman public key value*
2393 *ulPrivateDataLen* *the length in bytes of the second X9.42 Diffie-Hellman private key*
2394 *hPrivateData* *key handle for second X9.42 Diffie-Hellman private key value*
2395 *ulPublicDataLen2* *the length in bytes of the other party's second X9.42 Diffie-Hellman public key*
2396
2397 *pPublicData2* *pointer to other party's second X9.42 Diffie-Hellman public key value*
2398
2399 *publicKey* *Handle to the first party's ephemeral public key*

2400 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
2401 With the key derivation function **CKD_SHA1_KDF ASN1**, *pOtherInfo* must be supplied, which contains
2402 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
2403 the two parties intending to share the shared secret. With the key derivation function
2404 **CKD_SHA1_KDF CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
2405 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
2406 NULL and *ulOtherInfoLen* must be zero.

2407 **CK_X9_42_MQV_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_MQV_DERIVE_PARAMS**.

2408 2.4.12 X9.42 Diffie-Hellman key pair generation

2409 The X9.42 Diffie-Hellman key pair generation mechanism, denoted **CKM_X9_42_DH_KEY_PAIR_GEN**,
2410 is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in the ANSI
2411 X9.42 standard.

2412 It does not have a parameter.

2413 The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and
2414 subprime, as specified in the **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attributes of the template
2415 for the public key.

2416 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
2417 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, **CKA_SUBPRIME**, and
2418 **CKA_VALUE** attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman
2419 public and private key types must be specified in the templates.

2420 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2421 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

2422 **2.4.13 X9.42 Diffie-Hellman domain parameter generation**

2423 The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted
2424 **CKM_X9_42_DH_PARAMETER_GEN**, is a domain parameters generation mechanism based on X9.42
2425 Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.

2426 It does not have a parameter.

2427 The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime
2428 length in bits, as specified in the **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes of the
2429 template for the domain parameters.

2430 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**,
2431 **CKA_SUBPRIME**, **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes to the new object. Other
2432 attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the
2433 template for the domain parameters, or else are assigned default initial values.

2434 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2435 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits.

2436 **2.4.14 X9.42 Diffie-Hellman key derivation**

2437 The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM_X9_42_DH_DERIVE**, is a
2438 mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the
2439 ANSI X9.42 standard, where each party contributes one key pair, all using the same X9.42 Diffie-Hellman
2440 domain parameters.

2441 It has a parameter, a **CK_X9_42_DH1_DERIVE_PARAMS** structure.

2442 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
2443 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
2444 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
2445 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
2446 type must be specified in the template. Note that in order to validate this mechanism it may be required to
2447 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
2448 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

2449 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
2450 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
2451 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
2453 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
2454 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
2455 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
2457 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
2458 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
2459 value from its **CKA_EXTRACTABLE** attribute.

2461 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2462 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

2463 **2.4.15 X9.42 Diffie-Hellman hybrid key derivation**

2464 The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted
2465 **CKM_X9_42_DH_HYBRID_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman
2466 hybrid key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes two
2467 key pair, all using the same X9.42 Diffie-Hellman domain parameters.

2468 It has a parameter, a **CK_X9_42_DH2_DERIVE_PARAMS** structure.

2469 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
2470 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
2471 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
2472 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
2473 type must be specified in the template. Note that in order to validate this mechanism it may be required to
2474 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
2475 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

2476 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
CKA_SENSITIVE attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
value from its **CKA_EXTRACTABLE** attribute.

2488 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2489 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

2490 2.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

2491 The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted
2492 **CKM_X9_42_MQV_DERIVE**, is a mechanism for key derivation based the MQV scheme, as defined in
2493 the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-
2494 Hellman domain parameters.

2495 It has a parameter, a **CK_X9_42_MQV_DERIVE_PARAMS** structure.

2496 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
2497 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
2498 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
2499 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
2500 type must be specified in the template. Note that in order to validate this mechanism it may be required to
2501 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
2502 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

2503 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
CKA_SENSITIVE attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
value from its **CKA_EXTRACTABLE** attribute.

2515 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2516 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

2517 2.5 Extended Triple Diffie-Hellman (x3dh)

2518 The Extended Triple Diffie-Hellman mechanism described here is the one described in
2519 [SIGNAL].

2520

2521 *Table 61, Extended Triple Diffie-Hellman Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_X3DH_INITIATE							✓
CKM_X3DH RESPOND							✓

2522 2.5.1 Definitions

2523 Mechanisms:

2524 CKM_X3DH_INITIATE

2525 CKM_X3DH_RESPOND

2526 2.5.2 Extended Triple Diffie-Hellman key objects

2527 Extended Triple Diffie-Hellman uses Elliptic Curve keys in Montgomery representation
2528 (**CKK_EC_MONTGOMERY**). Three different kinds of keys are used, they differ in their lifespan:

- 2529 • identity keys are long-term keys, which identify the peer,
2530 • prekeys are short-term keys, which should be rotated often (weekly to hourly)
2531 • onetime prekeys are keys, which should be used only once.

2532 Any peer intending to be contacted using X3DH must publish their so-called prekey-bundle, consisting of
2533 their:

- 2534 • public Identity key,
2535 • current prekey, signed using XEDDA with their identity key
2536 • optionally a batch of One-time public keys.

2537 2.5.3 Initiating an Extended Triple Diffie-Hellman key exchange

2538 Initiating an Extended Triple Diffie-Hellman key exchange starts by retrieving the following required public
2539 keys (the so-called prekey-bundle) of the other peer: the Identity key, the signed public Prekey, and
2540 optionally one One-time public key.

2541 When the necessary key material is available, the initiating party calls
2542 CKM_X3DH_INITIATE, also providing the following additional parameters:

- 2543 • the initiators identity key
2544 • the initiators ephemeral key (a fresh, one-time **CKK_EC_MONTGOMERY** type key)

2545

2546 **CK_X3DH_INITIATE_PARAMS** is a structure that provides the parameters to the **CKM_X3DH_INITIATE**
2547 key exchange mechanism. The structure is defined as follows:

```
2548     typedef struct CK_X3DH_INITIATE_PARAMS {  
2549         CK_X3DH_KDF_TYPE      kdf;  
2550         CK_OBJECT_HANDLE     pPeer_identity;
```

```

2551     CK_OBJECT_HANDLE pPeer_prekey;
2552     CK_BYTE_PTR pPrekey_signature;
2553     CK_BYTE_PTR pOnetime_key;
2554     CK_OBJECT_HANDLE pOwn_identity;
2555     CK_OBJECT_HANDLE pOwn_ephemeral;
2556 } CK_X3DH_INITIATE_PARAMS;

```

2557 *Table 62, Extended Triple Diffie-Hellman Initiate Message parameters:*

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	<i>Key derivation function</i>
pPeer_identity	Key handle	<i>Peers public Identity key (from the prekey-bundle)</i>
pPeer_prekey	Key Handle	Peers public prekey (from the prekey-bundle)
pPrekey_signature	Byte array	<i>XEDDSA signature of PEER_PREKEY (from prekey-bundle)</i>
pOnetime_key	Byte array	Optional one-time public prekey of peer (from the prekey-bundle)
pOwn_identity	Key Handle	Initiators Identity key
pOwn_ephemeral	Key Handle	Initiators ephemeral key

2558

2559 **2.5.4 Responding to an Extended Triple Diffie-Hellman key exchange**

2560 Responding an Extended Triple Diffie-Hellman key exchange is done by executing a
2561 CKM_X3DH_RESPOND mechanism. **CK_X3DH_RESPOND_PARAMS** is a structure that provides the
2562 parameters to the **CKM_X3DH_RESPOND** key exchange mechanism. All these parameter should be
2563 supplied by the Initiator in a message to the responder. The structure is defined as follows:

```

2564     typedef struct CK_X3DH_RESPOND_PARAMS {
2565         CK_X3DH_KDF_TYPE kdf;
2566         CK_BYTE_PTR pIdentity_id;
2567         CK_BYTE_PTR pPrekey_id;
2568         CK_BYTE_PTR pOnetime_id;
2569         CK_OBJECT_HANDLE pInitiator_identity;
2570         CK_BYTE_PTR pInitiator_ephemeral;
2571     } CK_X3DH_RESPOND_PARAMS;

```

2572

2573 *Table 63, Extended Triple Diffie-Hellman 1st Message parameters:*

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	<i>Key derivation function</i>
pIdentity_id	Byte array	<i>Peers public Identity key identifier (from the prekey-bundle)</i>
pPrekey_id	Byte array	Peers public prekey identifier (from the prekey-bundle)
pOnetime_id	Byte array	Optional one-time public prekey of peer (from the prekey-bundle)
pInitiator_identity	Key handle	Initiators Identity key
pInitiator_ephemeral	Byte array	Initiators ephemeral key

2574

2575 Where the *_id fields are identifiers marking which key has been used from the prekey-bundle, these
2576 identifiers could be the keys themselves.

2577

2578 This mechanism has the following rules about key sensitivity and extractability³:

- 2579 1 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
2580 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
2581 default value.
- 2582 2 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
2583 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
2584 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
2585 **CKA_SENSITIVE** attribute.
- 2586 3 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
2587 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
2588 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
2589 value from its **CKA_EXTRACTABLE** attribute.

2590 **2.5.5 Extended Triple Diffie-Hellman parameters**

- 2591 • **CK_X3DH_KDF_TYPE, CK_X3DH_KDF_TYPE_PTR**

2592 **CK_X3DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying
2593 data from a shared secret. The key derivation function will be used by the X3DH key agreement
2594 schemes. It is defined as follows:

2595 **typedef CK ULONG CK_X3DH_KDF_TYPE;**

2596

2597 The following table lists the defined functions.

2598 *Table 64, X3DH: Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_512_KDF
CKD_SHA3_256_KDF
CKD_SHA256_KDF
CKD_SHA3_512_KDF
CKD_SHA512_KDF

2599 **2.6 Double Ratchet**

2600 The Double Ratchet is a key management algorithm managing the ongoing renewal and maintenance of
2601 short-lived session keys providing forward secrecy and break-in recovery for encrypt/decrypt operations.
2602 The algorithm is described in [DoubleRatchet]. The Signal protocol uses X3DH to exchange a shared
2603 secret in the first step, which is then used to derive a Double Ratchet secret key.

2604 *Table 65, Double Ratchet Mechanisms vs. Functions*

3 Note that the rules regarding the CKA_SENSITIVE, CKA_EXTRACTABLE, CKA_ALWAYS_SENSITIVE, and CKA_NEVER_EXTRACTABLE attributes have changed in version 2.11 to match the policy used by other key derivation mechanisms such as CKM_SSL3_MASTER_KEY_DERIVE.

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_X2RATCHET_INITIALIZE							✓
CKM_X2RATCHET_RESPOND							✓
CKM_X2RATCHET_ENCRYPT	✓					✓	
CKM_X2RATCHET_DECRYPT	✓					✓	

2605

2606 2.6.1 Definitions

2607 This section defines the key type "CKK_X2RATCHET" for type CK_KEY_TYPE as used in the
 2608 CKA_KEY_TYPE attribute of key objects.

2609 Mechanisms:

2610 CKM_X2RATCHET_INITIALIZE
 2611 CKM_X2RATCHET_RESPOND
 2612 CKM_X2RATCHET_ENCRYPT
 2613 CKM_X2RATCHET_DECRYPT

2614 2.6.2 Double Ratchet secret key objects

2615 Double Ratchet secret key objects (object class CKO_SECRET_KEY, key type CKK_X2RATCHET) hold
 2616 Double Ratchet keys. Double Ratchet secret keys can only be derived from shared secret keys using the
 2617 mechanism CKM_X2RATCHET_INITIALIZE or CKM_X2RATCHET_RESPOND. In the Signal protocol
 2618 these are seeded with the shared secret derived from an Extended Triple Diffie-Hellman [X3DH] key-
 2619 exchange. The following table defines the Double Ratchet secret key object attributes, in addition to the
 2620 common attributes defined for this object class:

2621 Table 66, Double Ratchet Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_X2RATCHET_RK	Byte array	Root key
CKA_X2RATCHET_HKS	Byte array	Sender Header key
CKA_X2RATCHET_HKR	Byte array	Receiver Header key
CKA_X2RATCHET_NHKR	Byte array	Next Sender Header Key
CKA_X2RATCHET_NHKR	Byte array	Next Receiver Header Key
CKA_X2RATCHET_CKS	Byte array	Sender Chain key
CKA_X2RATCHET_CKR	Byte array	Receiver Chain key
CKA_X2RATCHET_DHS	Byte array	Sender DH secret key
CKA_X2RATCHET_DHP	Byte array	Sender DH public key
CKA_X2RATCHET_DHR	Byte array	Receiver DH public key
CKA_X2RATCHET_NS	ULONG	Message number send
CKA_X2RATCHET_NR	ULONG	Message number receive
CKA_X2RATCHET_PNS	ULONG	Previous message number send
CKA_X2RATCHET_BOBS1STMSG	BOOL	Is this bob and has he ever sent a message?
CKA_X2RATCHET_ISALICE	BOOL	Is this Alice?
CKA_X2RATCHET_BAGSIZE	ULONG	How many out-of-order keys do we store
CKA_X2RATCHET_BAG	Byte array	Out-of-order keys

2622 2.6.3 Double Ratchet key derivation

2623 The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the
 2624 receiving, denoted **CKM_X2RATCHET_INITIALIZE** and **CKM_X2RATCHET RESPOND**, are the key
 2625 derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by
 2626 executing a X3DH key exchange.

2627 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 2628 key. Additionally the attribute flags indicating which functions the key supports are also contributed by the
 2629 mechanism.

2630 For this mechanism, the only allowed values are 255 and 448 as RFC 8032 only defines curves of these
 2631 two sizes. A Cryptoki implementation may support one or both of these curves and should set the
 2632 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

- 2633 • **CK_X2RATCHET_INITIALIZE_PARAMS;**
 2634 **CK_X2RATCHET_INITIALIZE_PARAMS_PTR**

2635 **CK_X2RATCHET_INITIALIZE_PARAMS** provides the parameters to the
 2636 **CKM_X2RATCHET_INITIALIZE** mechanism. It is defined as follows:

```
2637     typedef struct CK_X2RATCHET_INITIALIZE_PARAMS {
2638         CK_BYTE_PTR                         sk;
2639         CK_OBJECT_HANDLE                   peer_public_prekey;
2640         CK_OBJECT_HANDLE                   peer_public_identity;
2641         CK_OBJECT_HANDLE                   own_public_identity;
2642         CK_BBOOL                           bEncryptedHeader;
2643         CK ULONG                          eCurve;
2644         CK_MECHANISM_TYPE                aeadMechanism;
2645         CK_X2RATCHET_KDF_TYPE            kdfMechanism;
2646     } CK_X2RATCHET_INITIALIZE_PARAMS;
```

2647

2648 The fields of the structure have the following meanings:

2649 *sk* *the shared secret with peer (derived using X3DH)*

2650 *peers_public_prekey* *Peers public prekey which the Initiator used in the X3DH*

2651 *peers_public_identity* *Peers public identity which the Initiator used in the X3DH*

2652 *own_public_identity* *Initiators public identity as used in the X3DH*

2653 *bEncryptedHeader* *whether the headers are encrypted*

2654 *eCurve* *255 for curve 25519 or 448 for curve 448*

2655 *aeadMechanism* *a mechanism supporting AEAD encryption, e.g.*
 2656 *CKM_XCHACHA20*

2657 *kdfMechanism* *a Key Derivation Mechanism, such as*
 2658 *CKD_BLAKE2B_512_KDF*

- 2659 • **CK_X2RATCHET_RESPOND_PARAMS;**
- 2660 **CK_X2RATCHET_RESPOND_PARAMS_PTR**

2661 **CK_X2RATCHET_RESPOND_PARAMS** provides the parameters to the
 2662 **CKM_X2RATCHET_RESPOND** mechanism. It is defined as follows:

```
2663     typedef struct CK_X2RATCHET_RESPOND_PARAMS {
2664         CK_BYTE_PTR                                sk;
2665         CK_OBJECT_HANDLE                          own_prekey;
2666         CK_OBJECT_HANDLE                          initiator_identity;
2667         CK_OBJECT_HANDLE                          own_public_identity;
2668         CK_BBOOL                                  bEncryptedHeader;
2669         CK ULONG                                  eCurve;
2670         CK_MECHANISM_TYPE                        aeadMechanism;
2671         CK_X2RATCHET_KDF_TYPE                    kdfMechanism;
2672     } CK_X2RATCHET_RESPOND_PARAMS;
```

2673 The fields of the structure have the following meanings:

2674 *sk* *shared secret with the Initiator*

2675 *own_prekey* *Own Prekey pair that the Initiator used*

2676 *initiator_identity* *Initiators public identity key used*

2677 *own_public_identity* *as used in the prekey bundle by the initiator in the X3DH*

2678 *bEncryptedHeader* *whether the headers are encrypted*

2679 *eCurve* *255 for curve 25519 or 448 for curve 448*

2681 *aeadMechanism* *a mechanism supporting AEAD encryption, e.g.*
2682 *CKM_XCHACHA20*

2683 *kdfMechanism* *a Key Derivation Mechanism, such as*
2684 *CKD_BLAKE2B_512_KDF*

2685 **2.6.4 Double Ratchet Encryption mechanism**

2686 The Double Ratchet encryption mechanism, denoted **CKM_X2RATCHET_ENCRYPT** and
2687 **CKM_X2RATCHET_DECRYPT**, are a mechanisms for single part encryption and decryption based on
2688 the Double Ratchet and its underlying AEAD cipher.

2689 **2.6.5 Double Ratchet parameters**

- 2690 • **CK_X2RATCHET_KDF_TYPE, CK_X2RATCHET_KDF_TYPE_PTR**

2691 **CK_X2RATCHET_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive
2692 keying data from a shared secret. The key derivation function will be used by the X key derivation
2693 scheme. It is defined as follows:

```
2694        typedef CK ULONG CK_X2RATCHET_KDF_TYPE;
```

2695

2696 The following table lists the defined functions.

2697 *Table 67, X2RATCHET: Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_512_KDF
CKD_SHA3_256_KDF
CKD_SHA256_KDF
CKD_SHA3_512_KDF
CKD_SHA512_KDF

2698

2699 **2.7 Wrapping/unwrapping private keys**

2700 Cryptoki Versions 2.01 and up allow the use of secret keys for wrapping and unwrapping RSA private
2701 keys, Diffie-Hellman private keys, X9.42 Diffie-Hellman private keys, EC (also related to ECDSA) private
2702 keys and DSA private keys. For wrapping, a private key is BER-encoded according to PKCS #8's
2703 PrivateKeyInfo ASN.1 type. PKCS #8 requires an algorithm identifier for the type of the private key. The
2704 object identifiers for the required algorithm identifiers are as follows:

```
2705        rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }  
2706  
2707        dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }  
2708  
2709        dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2)  
2710                us(840) ansi-x942(10046) number-type(2) 1 }  
2711  
2712        id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2)
```

```

2713         us(840) ansi-x9-62(10045) publicKeyType(2) 1 }
2714
2715     id-dsa OBJECT IDENTIFIER ::= {
2716         iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }
2717
2718     where
2719     pkcs-1 OBJECT IDENTIFIER ::= {
2720         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }
2721
2722     pkcs-3 OBJECT IDENTIFIER ::= {
2723         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }
2724
2725     These parameters for the algorithm identifiers have the
2726             following types, respectively:
2727     NULL
2728
2729     DHParameter ::= SEQUENCE {
2730         prime           INTEGER,    -- p
2731         base            INTEGER,    -- g
2732         privateValueLength  INTEGER OPTIONAL
2733     }
2734
2735     DomainParameters ::= SEQUENCE {
2736         prime           INTEGER,    -- p
2737         base            INTEGER,    -- g
2738         subprime        INTEGER,    -- q
2739         cofactor        INTEGER OPTIONAL,   -- j
2740         validationParms ValidationParms OPTIONAL
2741     }
2742
2743     ValidationParms ::= SEQUENCE {
2744         Seed            BIT STRING, -- seed
2745         PGenCounter     INTEGER      -- parameter verification
2746     }
2747
2748     Parameters ::= CHOICE {
2749         ecParameters    ECParameters,
2750         namedCurve      CURVES.&id({CurveNames}),
2751         implicitlyCA   NULL
2752     }
2753
2754     Dss_Parms ::= SEQUENCE {
2755         p INTEGER,
2756         q INTEGER,
2757         g INTEGER
2758     }
2759

```

2760 For the X9.42 Diffie-Hellman domain parameters, the **cofactor** and the **validationParms** optional fields
2761 should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values
2762 are not stored within the token.

2763 For the EC domain parameters, the use of **namedCurve** is recommended over the choice
2764 **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

2765 Within the PrivateKeyInfo type:

- RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type requires values to be present for *all* the attributes specific to Cryptoki's RSA private key objects. In other words, if a Cryptoki library does not have values for an RSA private key's **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT2**, and **CKA_COEFFICIENT** values, it must not create an RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.
- Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
- X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
- EC (also related with ECDSA) private keys are BER-encoded according to SEC G SEC 1 ECPrivateKey ASN.1 type:

```
2766     ECPrivateKey ::= SEQUENCE {
2767         Version      INTEGER { ecPrivkeyVer1(1) }
2768         (ecPrivkeyVer1),
2769         privateKey   OCTET STRING,
2770         parameters  [0] Parameters OPTIONAL,
2771         publicKey    [1] BIT STRING OPTIONAL
2772     }
```

2783

2784 Since the EC domain parameters are placed in the PKCS #8's privateKeyAlgorithm field, the optional
2785 **parameters** field in an ECPrivateKey must be omitted. A Cryptoki application must be able to
2786 unwrap an ECPrivateKey that contains the optional **publicKey** field; however, what is done with this
2787 **publicKey** field is outside the scope of Cryptoki.

- DSA private keys are represented as BER-encoded ASN.1 type INTEGER.

2789 Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is
2790 encrypted with the secret key. This encryption must be done in CBC mode with PKCS padding.

2791 Unwrapping a wrapped private key undoes the above procedure. The CBC-encrypted ciphertext is
2792 decrypted, and the PKCS padding is removed. The data thereby obtained are parsed as a
2793 PrivateKeyInfo type, and the wrapped key is produced. An error will result if the original wrapped key
2794 does not decrypt properly, or if the decrypted unpadded data does not parse properly, or its type does not
2795 match the key type specified in the template for the new key. The unwrapping mechanism contributes
2796 only those attributes specified in the PrivateKeyInfo type to the newly-unwrapped key; other attributes
2797 must be specified in the template, or will take their default values.

2798 Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

```
2799     DSA OBJECT IDENTIFIER ::= { algorithm 12 }
2800     algorithm OBJECT IDENTIFIER ::= {
2801         iso(1) identifier-organization(3) oiw(14) secsig(3)
2802             algorithm(2) }
```

2803

2804 with associated parameters

```
2805     DSAParameters ::= SEQUENCE {
2806         prime1 INTEGER, -- modulus p
2807         prime2 INTEGER, -- modulus q
```

```

2808      base INTEGER -- base g
2809  }
2810
2811 for wrapping DSA private keys. Note that although the two structures for holding DSA domain
2812 parameters appear identical when instances of them are encoded, the two corresponding object
2813 identifiers are different.

```

2.8 Generic secret key

Table 68, Generic Secret Key Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GENERIC_SECRET_KEY_GEN					✓		

2.8.1 Definitions

This section defines the key type “CKK_GENERIC_SECRET” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_GENERIC_SECRET_KEY_GEN

2.8.2 Generic secret key objects

Generic secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_GENERIC_SECRET**) hold generic secret keys. These keys do not support encryption or decryption; however, other keys can be derived from them and they can be used in HMAC operations. The following table defines the generic secret key object attributes, in addition to the common attributes defined for this object class:

These key types are used in several of the mechanisms described in this section.

Table 69, Generic Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (arbitrary length)
CKA_VALUE_LEN ^{2,3}	CK ULONG	Length in bytes of key value

- Refer to [PKCS11-Base] table 11 for footnotes

The following is a sample template for creating a generic secret key object:

```

2830     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
2831     CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
2832     CK_UTF8CHAR label[] = "A generic secret key object";
2833     CK_BYTE value[] = { ... };
2834     CK_BBOOL true = CK_TRUE;
2835     CK_ATTRIBUTE template[] = {
2836         {CKA_CLASS, &class, sizeof(class)},
2837         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},

```

```
2838     { CKA_TOKEN, &true, sizeof(true) },
2839     { CKA_LABEL, label, sizeof(label)-1 },
2840     { CKA_DERIVE, &true, sizeof(true) },
2841     { CKA_VALUE, value, sizeof(value) }
2842 };
```

2843

2844 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three bytes of the SHA-1 hash of the generic secret key object's CKA_VALUE attribute.

2846 2.8.3 Generic secret key generation

2847 The generic secret key generation mechanism, denoted **CKM_GENERIC_SECRET_KEY_GEN**, is used
2848 to generate generic secret keys. The generated keys take on any attributes provided in the template
2849 passed to the **C_GenerateKey** call, and the **CKA_VALUE_LEN** attribute specifies the length of the key
2850 to be generated.

2851 It does not have a parameter.

2852 The template supplied must specify a value for the **CKA_VALUE_LEN** attribute. If the template specifies
2853 an object type and a class, they must have the following values:

```
2854     CK_OBJECT_CLASS = CKO_SECRET_KEY;
2855     CK_KEY_TYPE = CKK_GENERIC_SECRET;
```

2856 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2857 specify the supported range of key sizes, in bits.

2858 2.9 HMAC mechanisms

2859 Refer to **RFC2104** and **FIPS 198** for HMAC algorithm description. The HMAC secret key shall correspond
2860 to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism definition).
2861 Such keys, for use with HMAC operations can be created using **C_CreateObject** or **C_GenerateKey**.

2862 The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in
2863 the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

2864 2.9.1 General block cipher mechanism parameters

2865 • **CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR**

2866 **CK_MAC_GENERAL_PARAMS** provides the parameters to the general-length MACing mechanisms of
2867 the DES, DES3 (triple-DES), AES, Camellia, SEED, and ARIA ciphers. It also provides the parameters to
2868 the general-length HMACing mechanisms (i.e., SHA-1, SHA-256, SHA-384, SHA-512, and SHA-512/T
2869 family) and the two SSL 3.0 MACing mechanisms, (i.e., MD5 and SHA-1). It holds the length of the MAC
2870 that these mechanisms produce. It is defined as follows:

```
2871     typedef CK ULONG CK_MAC_GENERAL_PARAMS;
```

2872

2873 **CK_MAC_GENERAL_PARAMS_PTR** is a pointer to a **CK_MAC_GENERAL_PARAMS**.

2874 2.10 AES

2875 For the Advanced Encryption Standard (AES) see [FIPS PUB 197].

2876 *Table 70, AES Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_KEY_GEN					✓		
CKM_AES_ECB	✓					✓	
CKM_AES_CBC	✓					✓	
CKM_AES_CBC_PAD	✓					✓	
CKM_AES_MAC_GENERAL		✓					
CKM_AES_MAC		✓					
CKM_AES_OFB	✓					✓	
CKM_AES_CFB64	✓					✓	
CKM_AES_CFB8	✓					✓	
CKM_AES_CFB128	✓					✓	
CKM_AES_CFB1	✓					✓	
CKM_AES_XCBC_MAC		✓					
CKM_AES_XCBC_MAC_96		✓					

2.10.1 Definitions

This section defines the key type “CKK_AES” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

```

2881      CKM_AES_KEY_GEN
2882      CKM_AES_ECB
2883      CKM_AES_CBC
2884      CKM_AES_MAC
2885      CKM_AES_MAC_GENERAL
2886      CKM_AES_CBC_PAD
2887      CKM_AES_OFB
2888      CKM_AES_CFB64
2889      CKM_AES_CFB8
2890      CKM_AES_CFB128
2891      CKM_AES_CFB1
2892      CKM_AES_XCBC_MAC
2893      CKM_AES_XCBC_MAC_96

```

2.10.2 AES secret key objects

AES secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_AES**) hold AES keys. The following table defines the AES secret key object attributes, in addition to the common attributes defined for this object class:

2898 Table 71, AES Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN ^{2,3,6}	CK ULONG	Length in bytes of key value

- Refer to [PKCS11-Base] table 11 for footnotes

2900 The following is a sample template for creating an AES secret key object:

```
2901     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
2902     CK_KEY_TYPE keyType = CKK_AES;
2903     CK_UTF8CHAR label[] = "An AES secret key object";
2904     CK_BYTE value[] = {...};
2905     CK_BBOOL true = CK_TRUE;
2906     CK_ATTRIBUTE template[] = {
2907         {CKA_CLASS, &class, sizeof(class)},
2908         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
2909         {CKA_TOKEN, &true, sizeof(true)},
2910         {CKA_LABEL, label, sizeof(label)-1},
2911         {CKA_ENCRYPT, &true, sizeof(true)},
2912         {CKA_VALUE, value, sizeof(value)}}
2913     };
```

2914

2915 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
2916 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
2917 the key type of the secret key object.

2918 2.10.3 AES key generation

2919 The AES key generation mechanism, denoted **CKM_AES_KEY_GEN**, is a key generation mechanism for
2920 NIST's Advanced Encryption Standard.

2921 It does not have a parameter.

2922 The mechanism generates AES keys with a particular length in bytes, as specified in the
2923 **CKA_VALUE_LEN** attribute of the template for the key.

2924 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
2925 key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the
2926 key supports) may be specified in the template for the key, or else are assigned default initial values.

2927 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2928 specify the supported range of AES key sizes, in bytes.

2929 2.10.4 AES-ECB

2930 AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and
2931 decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and
2932 electronic codebook mode.

2933 It does not have a parameter.

2934 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
2935 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
2936 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
2937 one null bytes so that the resulting length is a multiple of the block size. The output data is the same

2938 length as the padded input data. It does not wrap the key type, key length, or any other information about
2939 the key; the application must convey these separately.

2940 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
2941 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
2942 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
2943 attribute of the new key; other attributes required by the key type must be specified in the template.

2944 Constraints on key types and the length of data are summarized in the following table:

2945 *Table 72, AES-ECB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	multiple of block size	same as input length	no final part
C_Decrypt	AES	multiple of block size	same as input length	no final part
C_WrapKey	AES	any	input length rounded up to multiple of block size	
C_UnwrapKey	AES	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

2946 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
2947 specify the supported range of AES key sizes, in bytes.

2948 2.10.5 AES-CBC

2949 AES-CBC, denoted **CKM_AES_CBC**, is a mechanism for single- and multiple-part encryption and
2950 decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard and
2951 cipher-block chaining mode.

2952 It has a parameter, a 16-byte initialization vector.

2953 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
2954 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
2955 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
2956 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
2957 length as the padded input data. It does not wrap the key type, key length, or any other information about
2958 the key; the application must convey these separately.

2959 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
2960 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
2961 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
2962 attribute of the new key; other attributes required by the key type must be specified in the template.

2963 Constraints on key types and the length of data are summarized in the following table:

2964 Table 73, AES-CBC: Key And Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	multiple of block size	same as input length	no final part
C_Decrypt	AES	multiple of block size	same as input length	no final part
C_WrapKey	AES	any	input length rounded up to multiple of the block size	
C_UnwrapKey	AES	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

2965 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 2966 specify the supported range of AES key sizes, in bytes.

2967 2.10.6 AES-CBC with PKCS padding

2968 AES-CBC with PKCS padding, denoted **CKM_AES_CBC_PAD**, is a mechanism for single- and multiple-
 2969 part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced
 2970 Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in PKCS
 2971 #7.

2972 It has a parameter, a 16-byte initialization vector.

2973 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
 2974 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified
 2975 for the **CKA_VALUE_LEN** attribute.

2976 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
 2977 Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section 2.7
 2978 for details). The entries in the table below for data length constraints when wrapping and unwrapping
 2979 keys do not apply to wrapping and unwrapping private keys.

2980 Constraints on key types and the length of data are summarized in the following table:

2981 Table 74, AES-CBC with PKCS Padding: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt	AES	any	input length rounded up to multiple of the block size
C_Decrypt	AES	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	AES	any	input length rounded up to multiple of the block size
C_UnwrapKey	AES	multiple of block size	between 1 and block length bytes shorter than input length

2982 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 2983 specify the supported range of AES key sizes, in bytes.

2984 2.10.7 AES-OFB

2985 AES-OFB, denoted **CKM_AES_OFB**. It is a mechanism for single and multiple-part encryption and
 2986 decryption with AES. AES-OFB mode is described in [NIST sp800-38a].

2987 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
 2988 the block size.

2989

2990 Constraints on key types and the length of data are summarized in the following table:

2991

2992 *Table 75, AES-OFB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	any	same as input length	no final part
C_Decrypt	AES	any	same as input length	no final part

2993 For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

2.10.8 AES-CFB

2995 Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM_AES_CFB8, CKM_AES_CFB64, and
2996 CKM_AES_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES.
2997 AES-OFB mode is described [NIST sp800-38a].

3000 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
3001 the block size.

3000

3001 Constraints on key types and the length of data are summarized in the following table:

3002

3003 *Table 76, AES-CFB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	any	same as input length	no final part
C_Decrypt	AES	any	same as input length	no final part

3004 For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

2.10.9 General-length AES-MAC

3006 General-length AES-MAC, denoted **CKM_AES_MAC_GENERAL**, is a mechanism for single- and
3007 multiple-part signatures and verification, based on NIST Advanced Encryption Standard as defined in
3008 FIPS PUB 197 and data authentication as defined in FIPS PUB 113.

3009 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
3010 desired from the mechanism.

3011 The output bytes from this mechanism are taken from the start of the final AES cipher block produced in
3012 the MACing process.

3013 Constraints on key types and the length of data are summarized in the following table:

3014 *Table 77, General-length AES-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	any	1-block size, as specified in parameters
C_Verify	AES	any	1-block size, as specified in parameters

3015 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3016 specify the supported range of AES key sizes, in bytes.

2.10.10 AES-MAC

3018 AES-MAC, denoted by **CKM_AES_MAC**, is a special case of the general-length AES-MAC mechanism.
3019 AES-MAC always produces and verifies MACs that are half the block size in length.

3020 It does not have a parameter.

3021 Constraints on key types and the length of data are summarized in the following table:

3022 *Table 78, AES-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	Any	½ block size (8 bytes)
C_Verify	AES	Any	½ block size (8 bytes)

3023 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3024 specify the supported range of AES key sizes, in bytes.

3025 **2.10.11 AES-XCBC-MAC**

3026 AES-XCBC-MAC, denoted **CKM_AES_XCBC_MAC**, is a mechanism for single and multiple part
3027 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

3028 It does not have a parameter.

3029 Constraints on key types and the length of data are summarized in the following table:

3030 *Table 79, AES-XCBC-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	Any	16 bytes
C_Verify	AES	Any	16 bytes

3031 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3032 specify the supported range of AES key sizes, in bytes.

3033 **2.10.12 AES-XCBC-MAC-96**

3034 AES-XCBC-MAC-96, denoted **CKM_AES_XCBC_MAC-96**, is a mechanism for single and multiple part
3035 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

3036 It does not have a parameter.

3037 Constraints on key types and the length of data are summarized in the following table:

3038 *Table 80, AES-XCBC-MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	AES	Any	12 bytes
C_Verify	AES	Any	12 bytes

3039 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3040 specify the supported range of AES key sizes, in bytes.

3041 **2.11 AES with Counter**

3042 *Table 81, AES with Counter Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CTR	✓					✓	

3043 **2.11.1 Definitions**

3044 Mechanisms:

3045 **CKM_AES_CTR**

3046 **2.11.2 AES with Counter mechanism parameters**

3047 ◆ **CK_AES_CTR_PARAMS; CK_AES_CTR_PARAMS_PTR**

3048 **CK_AES_CTR_PARAMS** is a structure that provides the parameters to the **CKM_AES_CTR** mechanism.
3049 It is defined as follows:

```
3050     typedef struct CK_AES_CTR_PARAMS {  
3051         CK ULONG ulCounterBits;  
3052         CK BYTE cb[16];  
3053     } CK_AES_CTR_PARAMS;  
3054
```

3055 ulCounterBits specifies the number of bits in the counter block (cb) that shall be incremented. This
3056 number shall be such that $0 < ulCounterBits \leq 128$. For any values outside this range the mechanism
3057 shall return **CKR_MECHANISM_PARAM_INVALID**.

3058 It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter
3059 bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting
3060 with 1. The rest of 'cb' is for the nonce, and maybe an optional IV.

3061 E.g. as defined in [RFC 3686]:

```
3062     0           1           2           3  
3063     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1  
3064     +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
3065     |           Nonce           |  
3066     +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
3067     |           Initialization Vector (IV)           |  
3068     |  
3069     +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
3070     |           Block Counter           |  
3071     +-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

3072
3073 This construction permits each packet to consist of up to $2^{32}-1$ blocks = 4,294,967,295 blocks =
3074 68,719,476,720 octets.

3075 **CK_AES_CTR_PARAMS_PTR** is a pointer to a **CK_AES_CTR_PARAMS**.

3076 **2.11.3 AES with Counter Encryption / Decryption**

3077 Generic AES counter mode is described in NIST Special Publication 800-38A and in RFC 3686. These
3078 describe encryption using a counter block which may include a nonce to guarantee uniqueness of the
3079 counter block. Since the nonce is not incremented, the mechanism parameter must specify the number of
3080 counter bits in the counter block.

3081 The block counter is incremented by 1 after each block of plaintext is processed. There is no support for
3082 any other increment functions in this mechanism.

3083 If an attempt to encrypt/decrypt is made which will cause an overflow of the counter block's counter bits,
3084 then the mechanism shall return **CKR_DATA_LEN_RANGE**. Note that the mechanism should allow the
3085 final post increment of the counter to overflow (if it implements it this way) but not allow any further
3086 processing after this point. E.g. if ulCounterBits = 2 and the counter bits start as 1 then only 3 blocks of
3087 data can be processed.

3088

3089 **2.12 AES CBC with Cipher Text Stealing CTS**

3090 Ref [NIST AES CTS]

3091 This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to
3092 the same length of cipher text.

3093 *Table 82, AES CBC with Cipher Text Stealing CTS Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CTS	✓					✓	

3094 **2.12.1 Definitions**

3095 Mechanisms:

3096 CKM_AES_CTS

3097 **2.12.2 AES CTS mechanism parameters**

3098 It has a parameter, a 16-byte initialization vector.

3099 *Table 83, AES-CTS: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	AES	Any, ≥ block size (16 bytes)	same as input length	no final part
C_Decrypt	AES	any, ≥ block size (16 bytes)	same as input length	no final part

3100

3101 **2.13 Additional AES Mechanisms**

3102 *Table 84, Additional AES Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_GCM	✓					✓	
CKM_AES_CCM	✓					✓	
CKM_AES_GMAC		✓					

3103

3104 **2.13.1 Definitions**

3105 Mechanisms:

3106 CKM_AES_GCM

3107 CKM_AES_CCM

3108 CKM_AES_GMAC

3109 Generator Functions:

3110 CKG_NO_GENERATE

3111 CKG_GENERATE
3112 CKG_GENERATE_COUNTER
3113 CKG_GENERATE_RANDOM

3114 2.13.2 AES-GCM Authenticated Encryption / Decryption

3115 Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where K
3116 (key) and AAD (additional authenticated data) are as described in [GCM]. AES-GCM uses
3117 CK_GCM_PARAMS for Encrypt, Decrypt and CK_GCM_MESSAGE_PARAMS for MessageEncrypt and
3118 MessageDecrypt.

3119 Encrypt:

- 3120 • Set the IV length $ullvLen$ in the parameter block.
- 3121 • Set the IV data p/v in the parameter block.
- 3122 • Set the AAD data $pAAD$ and size $ulAADLen$ in the parameter block. $pAAD$ may be NULL if
3123 $ulAADLen$ is 0.
- 3124 • Set the tag length $ulTagBits$ in the parameter block.
- 3125 • Call C_EncryptInit() for **CKM_AES_GCM** mechanism with parameters and key K .
- 3126 • Call C_Encrypt(), or C_EncryptUpdate()⁴ C_EncryptFinal(), for the plaintext obtaining ciphertext
3127 and authentication tag output.

3128 Decrypt:

- 3129 • Set the IV length $ullvLen$ in the parameter block.
- 3130 • Set the IV data p/v in the parameter block.
- 3131 • Set the AAD data $pAAD$ and size $ulAADLen$ in the parameter block. $pAAD$ may be NULL if
3132 $ulAADLen$ is 0.
- 3133 • Set the tag length $ulTagBits$ in the parameter block.
- 3134 • Call C_DecryptInit() for **CKM_AES_GCM** mechanism with parameters and key K .
- 3135 • Call C_Decrypt(), or C_DecryptUpdate()¹ C_DecryptFinal(), for the ciphertext, including the
3136 appended tag, obtaining plaintext output. Note: since **CKM_AES_GCM** is an AEAD cipher, no
3137 data should be returned until C_Decrypt() or C_DecryptFinal().

3138 MessageEncrypt:

- 3139 • Set the IV length $ullvLen$ in the parameter block.
- 3140 • Set p/v to hold the IV data returned from C_EncryptMessage() and C_EncryptMessageBegin(). If
3141 $ullvFixedBits$ is not zero, then the most significant bits of p/V contain the fixed IV. If $ivGenerator$ is
3142 set to CKG_NO_GENERATE, p/v is an input parameter with the full IV.
- 3143 • Set the $ullvFixedBits$ and $ivGenerator$ fields in the parameter block.
- 3144 • Set the tag length $ulTagBits$ in the parameter block.
- 3145 • Set $pTag$ to hold the tag data returned from C_EncryptMessage() or the final
3146 C_EncryptMessageNext().
- 3147 • Call C_MessageEncryptInit() for **CKM_AES_GCM** mechanism key K .

⁴ ** indicates 0 or more calls may be made as required

- 3148 • Call C_EncryptMessage(), or C_EncryptMessageBegin() followed by C_EncryptMessageNext()⁵.
 3149 The mechanism parameter is passed to all three of these functions.
 3150 • Call C_MessageEncryptFinal() to close the message decryption.

3151 MessageDecrypt:

- 3152 • Set the IV length *ullvLen* in the parameter block.
- 3153 • Set the IV data *pIV* in the parameter block.
- 3154 • The *ullvFixedBits* and *ivGenerator* fields are ignored.
- 3155 • Set the tag length *ulTagBits* in the parameter block.
- 3156 • Set the tag data *pTag* in the parameter block before C_DecryptMessage() or the final
 3157 C_DecryptMessageNext().
- 3158 • Call C_MessageDecryptInit() for **CKM_AES_GCM** mechanism key *K*.
- 3159 • Call C_DecryptMessage(), or C_DecryptMessageBegin followed by C_DecryptMessageNext()⁶.
 3160 The mechanism parameter is passed to all three of these functions.
- 3161 • Call C_MessageDecryptFinal() to close the message decryption.

3162 In *pIV* the least significant bit of the initialization vector is the rightmost bit. *ullvLen* is the length of the
 3163 initialization vector in bytes.

3164 On MessageEncrypt, the meaning of *ivGenerator* is as follows: CKG_NO_GENERATE means the IV is
 3165 passed in on MessageEncrypt and no internal IV generation is done. CKG_GENERATE means that the
 3166 non-fixed portion of the IV is generated by the module internally. The generation method is not defined.
 3167 CKG_GENERATE_COUNTER means that the non-fixed portion of the IV is generated by the module
 3168 internally by use of an incrementing counter. CKG_GENERATE_RANDOM means that the non-fixed
 3169 portion of the IV is generated by the module internally using a PRNG. In any case the entire IV, including
 3170 the fixed portion, is returned in *pIV*.

3171 Modules must implement CKG_GENERATE. Modules may also reject *ullvFixedBits* values which are too
 3172 large. Zero is always an acceptable value for *ullvFixedBits*.

3173 In Encrypt and Decrypt the tag is appended to the cipher text and the least significant bit of the tag is the
 3174 rightmost bit and the tag bits are the rightmost *ulTagBits* bits. In MessageEncrypt the tag is returned in
 3175 the *pTag* field of CK_GCM_MESSAGE_PARAMS. In MessageDecrypt the tag is provided by the *pTag*
 3176 field of CK_GCM_MESSAGE_PARAMS.

3177 The key type for *K* must be compatible with **CKM_AES_ECB** and the
 3178 C_EncryptInit()/C_DecryptInit()/C_MessageEncryptInit()/C_MessageDecryptInit() calls shall behave, with
 3179 respect to *K*, as if they were called directly with **CKM_AES_ECB**, *K* and NULL parameters.

3180 **2.13.3 AES-CCM authenticated Encryption / Decryption**

3181 For IPsec (RFC 4309) and also for use in ZFS encryption. Generic CCM mode is described in [RFC
 3182 3610].

3183 To set up for AES-CCM use the following process, where *K* (key), nonce and additional authenticated
 3184 data are as described in [RFC 3610]. AES-CCM uses CK_CCM_PARAMS for Encrypt and Decrypt, and
 3185 CK_CCM_MESSAGE_PARAMS for MessageEncrypt and MessageDecrypt.

3186 Encrypt:

- 3187 • Set the message/data length *ulDataLen* in the parameter block.

5 ^{**} indicates 0 or more calls may be made as required

6 ^{**} indicates 0 or more calls may be made as required

- 3188 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- 3189 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- 3191 • Set the MAC length *ulMACLen* in the parameter block.
- 3192 • Call *C_EncryptInit()* for **CKM_AES_CCM** mechanism with parameters and key *K*.
- 3193 • Call *C_Encrypt()*, *C_EncryptUpdate()*, or *C_EncryptFinal()*, for the plaintext obtaining the final ciphertext output and the MAC. The total length of data processed must be *ulDataLen*. The output length will be *ulDataLen* + *ulMACLen*.
- 3196 Decrypt:
- 3197 • Set the message/data length *ulDataLen* in the parameter block. This length must not include the length of the MAC that is appended to the cipher text.
- 3199 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- 3200 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- 3202 • Set the MAC length *ulMACLen* in the parameter block.
- 3203 • Call *C_DecryptInit()* for **CKM_AES_CCM** mechanism with parameters and key *K*.
- 3204 • Call *C_Decrypt()*, *C_DecryptUpdate()*, or *C_DecryptFinal()*, for the ciphertext, including the appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen* + *ulMACLen*. Note: since **CKM_AES_CCM** is an AEAD cipher, no data should be returned until *C_Decrypt()* or *C_DecryptFinal()*.
- 3208 MessageEncrypt:
- 3209 • Set the message/data length *ulDataLen* in the parameter block.
- 3210 • Set the nonce length *ulNonceLen*.
- 3211 • Set *pNonce* to hold the nonce data returned from *C_EncryptMessage()* and *C_EncryptMessageBegin()*. If *ulNonceFixedBits* is not zero, then the most significant bits of *pNonce* contain the fixed nonce. If *nonceGenerator* is set to **CKG_NO_GENERATE**, *pNonce* is an input parameter with the full nonce.
- 3215 • Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.
- 3216 • Set the MAC length *ulMACLen* in the parameter block.
- 3217 • Set *pMAC* to hold the MAC data returned from *C_EncryptMessage()* or the final *C_EncryptMessageNext()*.
- 3219 • Call *C_MessageEncryptInit()* for **CKM_AES_CCM** mechanism key *K*.
- 3220 • Call *C_EncryptMessage()*, or *C_EncryptMessageBegin()* followed by *C_EncryptMessageNext()*⁷. The mechanism parameter is passed to all three functions.
- 3222 • Call *C_MessageEncryptFinal()* to close the message encryption.
- 3223 • The MAC is returned in *pMac* of the **CK_CCM_MESSAGE_PARAMS** structure.
- 3224 MessageDecrypt:
- 3225 • Set the message/data length *ulDataLen* in the parameter block.

⁷ ^{**} indicates 0 or more calls may be made as required

- Set the nonce length *uNonceLen* and the nonce data *pNonce* in the parameter block
- The *uNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.
- Set the MAC length *uMACLen* in the parameter block.
- Set the MAC data *pMAC* in the parameter block before *C_DecryptMessage()* or the final *C_DecryptMessageNext()*.
- Call *C_MessageDecryptInit()* for **CKM_AES_CCM** mechanism key *K*.
- Call *C_DecryptMessage()*, or *C_DecryptMessageBegin()* followed by *C_DecryptMessageNext()*⁸. The mechanism parameter is passed to all three functions.
- Call *C_MessageDecryptFinal()* to close the message decryption.

In *pNonce* the least significant bit of the nonce is the rightmost bit. *uNonceLen* is the length of the nonce in bytes.

On *MessageEncrypt*, the meaning of *nonceGenerator* is as follows: **CKG_NO_GENERATE** means the nonce is passed in on *MessageEncrypt* and no internal MAC generation is done. **CKG_GENERATE** means that the non-fixed portion of the nonce is generated by the module internally. The generation method is not defined. **CKG_GENERATE_COUNTER** means that the non-fixed portion of the nonce is generated by the module internally by use of an incrementing counter. **CKG_GENERATE_RANDOM** means that the non-fixed portion of the nonce is generated by the module internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.

Modules must implement **CKG_GENERATE**. Modules may also reject *uNonceFixedBits* values which are too large. Zero is always an acceptable value for *uNonceFixedBits*.

In *Encrypt* and *Decrypt* the MAC is appended to the cipher text and the least significant byte of the MAC is the rightmost byte and the MAC bytes are the rightmost *uMACLen* bytes. In *MessageEncrypt* the MAC is returned in the *pMAC* field of *CK_CCM_MESSAGE_PARAMS*. In *MessageDecrypt* the MAC is provided by the *pMAC* field of *CK_CCM_MESSAGE_PARAMS*.

The key type for *K* must be compatible with **CKM_AES_ECB** and the *C_EncryptInit()*/*C_DecryptInit()*/*C_MessageEncryptInit()*/*C_MessageDecryptInit()* calls shall behave, with respect to *K*, as if they were called directly with **CKM_AES_ECB**, *K* and *NULL* parameters.

2.13.4 AES-GMAC

AES-GMAC, denoted **CKM_AES_GMAC**, is a mechanism for single and multiple-part signatures and verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism parameters. When GMAC is used with *C_Sign* or *C_Verify*, *pData* points to the AAD. GMAC does not use plaintext or ciphertext.

The signature produced by GMAC, also referred to as a Tag, the tag's length is determined by the *CK_GMAC_PARAMS* field *uTagBits*.

The IV length is determined by the *CK_GMAC_PARAMS* field *uIVLen*.

Constraints on key types and the length of data are summarized in the following table:

⁸ “*” indicates 0 or more calls may be made as required

3263 Table 85, AES-GMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	< 2^64	Depends on param's ulTagBits
C_Verify	CKK_AES	< 2^64	Depends on param's ulTagBits

3264 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
3265 specify the supported range of AES key sizes, in bytes.

3266 2.13.5 AES GCM and CCM Mechanism parameters

3267 ♦ CK_GENERATOR_FUNCTION

3268 Functions to generate unique IVs and nonces.

```
3269     typedef CK ULONG CK_GENERATOR_FUCNTION;
```

3270 ♦ CK_GCM_PARAMS; CK_GCM_PARAMS_PTR

3271 CK_GCM_PARAMS is a structure that provides the parameters to the CKM_AES_GCM mechanism
3272 when used for Encrypt or Decrypt. It is defined as follows:

```
3273     typedef struct CK_GCM_PARAMS {  
3274         CK_BYTE_PTR    pIV;  
3275         CK ULONG      ulIVLen;  
3276         CK_BYTE_PTR    pAAD;  
3277         CK ULONG      ulAADLen;  
3278         CK ULONG      ulTagBits;  
3279     } CK_GCM_PARAMS;
```

3280

3281 The fields of the structure have the following meanings:

3282 pIV pointer to initialization vector

3283 ulIVLen length of initialization vector in bytes. The length of the initialization
3284 vector can be any number between 1 and (2^32) - 1. 96-bit (12
3285 byte) IV values can be processed more efficiently, so that length is
3286 recommended for situations in which efficiency is critical.

3287 pAAD pointer to additional authentication data. This data is authenticated
3288 but not encrypted.

3289 ulAADLen length of pAAD in bytes. The length of the AAD can be any number
3290 between 0 and (2^32) – 1.

3291 ulTagBits length of authentication tag (output following cipher text) in bits. Can
3292 be any value between 0 and 128.

3293 CK_GCM_PARAMS_PTR is a pointer to a CK_GCM_PARAMS.

3294 ♦ CK_GCM_MESSAGE_PARAMS; CK_GCM_MESSAGE_PARAMS_PTR

3295 CK_GCM_MESSAGE_PARAMS is a structure that provides the parameters to the CKM_AES_GCM
3296 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```

3297     typedef struct CK_GCM_MESSAGE_PARAMS {
3298         CK_BYTE_PTR    pIv;
3299         CK ULONG       ulIvLen;
3300         CK ULONG       ulIvFixedBits;
3301         CK_GENERATOR_FUNCTION ivGenerator;
3302         CK_BYTE_PTR    pTag;
3303         CK ULONG       ulTagBits;
3304     } CK_GCM_MESSAGE_PARAMS;
3305
3306 The fields of the structure have the following meanings:
3307     plv      pointer to initialization vector
3308     ullvLen   length of initialization vector in bytes. The length of the initialization
3309                           vector can be any number between 1 and (2^32) - 1. 96-bit (12 byte)
3310                           IV values can be processed more efficiently, so that length is
3311                           recommended for situations in which efficiency is critical.
3312     ullvFixedBits  number of bits of the original IV to preserve when generating a new IV. These bits are counted from the Most significant bits (to the right).
3313
3314
3315     ivGenerator  Function used to generate a new IV. Each IV must be unique for a given session.
3316
3317     pTag        location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.
3318
3319     ulTagBits   length of authentication tag in bits. Can be any value between 0 and 128.
3320

```

3321 **CK_GCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_GCM_MESSAGE_PARAMS**.

3322

3323 ♦ **CK_CCM_PARAMS; CK_CCM_PARAMS_PTR**

3324 **CK_CCM_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism
3325 when used for Encrypt or Decrypt. It is defined as follows:

```

3326     typedef struct CK_CCM_PARAMS {
3327         CK ULONG       ulDataLen; /*plaintext or ciphertext*/
3328         CK_BYTE_PTR    pNonce;
3329         CK ULONG       ulNonceLen;
3330         CK_BYTE_PTR    pAAD;
3331         CK ULONG       ulAADLen;
3332         CK ULONG       ulMACLen;
3333     } CK_CCM_PARAMS;

```

3334 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
3335 length ($2 \leq L \leq 8$):

```

3336     ulDataLen   length of the data where  $0 \leq ulDataLen < 2^{8L}$ .
3337     pNonce      the nonce.

```

3338

- ulNonceLen* *length of pNonce in bytes where 7 <= ulNonceLen <= 13.*

3339

- pAAD* *Additional authentication data. This data is authenticated but not encrypted.*

3340

3341

- ulAADLen* *length of pAAD in bytes where 0 <= ulAADLen <= (2^32) - 1.*

3342

- ulMACLen* *length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.*

3343

3344 **CK_CCM_PARAMS_PTR** is a pointer to a **CK_CCM_PARAMS**.

3345 ◆ **CK_CCM_MESSAGE_PARAMS; CK_CCM_MESSAGE_PARAMS_PTR**

3346 **CK_CCM_MESSAGE_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```

3348        typedef struct CK_CCM_MESSAGE_PARAMS {
3349            CK ULONG        

- ulDataLen;   /*plaintext or ciphertext*/


3350            CK BYTE PTR    

- pNonce;


3351            CK ULONG        

- ulNonceLen;


3352            CK ULONG        

- ulNonceFixedBits;


3353            CK GENERATOR FUNCTION   

- nonceGenerator;


3354            CK BYTE PTR    

- pMAC;


3355            CK ULONG        

- ulMACLen;


3356        } CK_CCM_MESSAGE_PARAMS;
  
```

3357

3358 The fields of the structure have the following meanings, where L is the size in bytes of the data length's length ($2 \leq L \leq 8$):

3359

3360

- ulDataLen* *length of the data where 0 <= ulDataLen < 2^(8L).*

3361

- pNonce* *the nonce.*

3362

- ulNonceLen* *length of pNonce in bytes where 7 <= ulNonceLen <= 13.*

3363

- ulNonceFixedBits* *number of bits of the original nonce to preserve when generating a new nonce. These bits are counted from the Most significant bits (to the right).*

3364

3365

3366

- nonceGenerator* *Function used to generate a new nonce. Each nonce must be unique for a given session.*

3367

3368

- pMAC* *location of the CCM MAC returned on MessageEncrypt, provided on MessageDecrypt*

3369

3370

- ulMACLen* *length of the MAC (output following cipher text) in bytes. Valid values are 4, 6, 8, 10, 12, 14, and 16.*

3371

3372 **CK_CCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_CCM_MESSAGE_PARAMS**.

3373

3374 2.14 AES CMAC

3375 *Table 86, Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_CMAC_GENERAL		✓					
CKM_AES_CMAC		✓					

3376 ¹ SR = SignRecover, VR = VerifyRecover.

2.14.1 Definitions

3377 Mechanisms:

3379 CKM_AES_CMAC_GENERAL

3380 CKM_AES_CMAC

2.14.2 Mechanism parameters

3382 CKM_AES_CMAC_GENERAL uses the existing **CK_MAC_GENERAL_PARAMS** structure.

3383 CKM_AES_CMAC does not use a mechanism parameter.

2.14.3 General-length AES-CMAC

3385 General-length AES-CMAC, denoted **CKM_AES_CMAC_GENERAL**, is a mechanism for single- and
3386 multiple-part signatures and verification, based on [NIST SP800-38B] and [RFC 4493].

3387 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
3388 desired from the mechanism.

3389 The output bytes from this mechanism are taken from the start of the final AES cipher block produced in
3390 the MACing process.

3391 Constraints on key types and the length of data are summarized in the following table:

3392 *Table 87, General-length AES-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_AES	any	1-block size, as specified in parameters
C_Verify	CKK_AES	any	1-block size, as specified in parameters

3393 References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less
3394 than 64 bits. The MAC length must be specified before the communication starts, and must not be
3395 changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

3396 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3397 specify the supported range of AES key sizes, in bytes.

2.14.4 AES-CMAC

3399 AES-CMAC, denoted **CKM_AES_CMAC**, is a special case of the general-length AES-CMAC mechanism.
3400 AES-MAC always produces and verifies MACs that are a full block size in length, the default output length
3401 specified by [RFC 4493].

3402 Constraints on key types and the length of data are summarized in the following table:

3403 *Table 88, AES-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length	
C_Sign	CKK_AES	any	Block size (16 bytes)	
C_Verify	CKK_AES	any	Block size (16 bytes)	

3404 References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less
 3405 than 64 bits. The MAC length must be specified before the communication starts, and must not be
 3406 changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

3407 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 3408 specify the supported range of AES key sizes, in bytes.

3409 **2.15 AES XTS**

3410 *Table 89, Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_XTS	✓					✓	
CKM_AES_XTS_KEY_GEN					✓		

3411 **2.15.1 Definitions**

3412 This section defines the key type "CKK_AES_XTS" for type CK_KEY_TYPE as used in the
 3413 CKA_KEY_TYPE attribute of key objects.

3414 Mechanisms:

3415 CKM_AES_XTS
 3416 CKM_AES_CTS_KEY_GEN

3417 **2.15.2 AES-XTS secret key objects**

3418 *Table 90, AES-XTS Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (32 or 64 bytes)
CKA_VALUE_LEN ^{2,3,6}	CK ULONG	Length in bytes of key value

3419 - Refer to [PKCS11-Base] table 11 for footnotes

3420 **2.15.3 AES-XTS key generation**

3421 The double-length AES-XTS key generation mechanism, denoted **CKM_AES_XTS_KEY_GEN**, is a key
 3422 generation mechanism for double-length AES-XTS keys.

3423 The mechanism generates AES-XTS keys with a particular length in bytes as specified in the
 3424 CKA_VALUE_LEN attributes of the template for the key.

3425 This mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new
 3426 key. Other attributes supported by the double-length AES-XTS key type (specifically, the flags indicating
 3427 which functions the key supports) may be specified in the template for the key, or else are assigned
 3428 default initial values.

3429 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 3430 specify the supported range of AES-XTS key sizes, in bytes.

2.15.4 AES-XTS

AES-XTS (XEX-based Tweaked CodeBook mode with CipherText Stealing), denoted **CKM_AES_XTS**, is a mechanism for single- and multiple-part encryption and decryption. It is specified in NIST SP800-38E.

Its single parameter is a Data Unit Sequence Number 16 bytes long. Supported key lengths are 32 and 64 bytes. Keys are internally split into half-length sub-keys of 16 and 32 bytes respectively. Constraints on key types and the length of data are summarized in the following table:

Table 91, AES-XTS: Key And Data Length

Function	Key type	Input length	Output length			Comments
C_Encrypt	CKK_AES_XTS	Any, ≥ block size (16 bytes)	Same as input length			No final part
C_Decrypt	CKK_AES_XTS	Any, ≥ block size (16 bytes)	Same as input length			No final part

3438

2.16 AES Key Wrap

Table 92, AES Key Wrap Mechanisms vs. Functions

3441

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_AES_KEY_WRAP	✓					✓	
CKM_AES_KEY_WRAP_PAD	✓					✓	
CKM_AES_KEY_WRAP_KWP	✓					✓	

¹SR = SignRecover, VR = VerifyRecover

2.16.1 Definitions

Mechanisms:

CKM_AES_KEY_WRAP

CKM_AES_KEY_WRAP_PAD

CKM_AES_KEY_WRAP_KWP

2.16.2 AES Key Wrap Mechanism parameters

The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if present, must be a fixed size array of 8 bytes for CKM_AES_KEY_WRAP and CKM_AES_KEY_WRAP_PAD, resp. 4 bytes for CKM_AES_KEY_WRAP_KWP; and, if NULL, will use the default initial value defined in Section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

The type of this parameter is CK_BYTE_PTR and the pointer points to the array of bytes to be used as the initial value. The length shall be either 0 and the pointer NULL; or 8 for CKM_AES_KEY_WRAP / CKM_AES_KEY_WRAP_PAD, resp. 4 for CKM_AES_KEY_WRAP_KWP, and the pointer non-NUL.

2.16.3 AES Key Wrap

The mechanisms support only single-part operations, single part wrapping and unwrapping, and single-part encryption and decryption.

3458 The CKM_AES_KEY_WRAP mechanism can only wrap a key resp. encrypt a block of data whose size is
 3459 an exact multiple of the AES Key Wrap algorithm block size. Wrapping / encryption is done as defined in
 3460 Section 6.2 of [AES KEYWRAP].

3461 The CKM_AES_KEY_WRAP_PAD mechanism can wrap a key or encrypt a block of data of any length. It
 3462 does the padding detailed in PKCS #7 of inputs (keys or data blocks), always producing wrapped output
 3463 that is larger than the input key/data to be wrapped. This padding is done by the token before being
 3464 passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of data as defined
 3465 in Section 6.2 of [AES KEYWRAP].

3466 The CKM_AES_KEY_WRAP_KWP mechanism can wrap a key or encrypt block of data of any length.
 3467 The input is padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which
 3468 produces same results as RFC 5649.

3469 2.17 Key derivation by data encryption – DES & AES

3470 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
 3471 They are for use with the C_DeriveKey function.

3472 *Table 93, Key derivation by data encryption Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES_ECB_ENCRYPT_DATA							✓
CKM_DES_CBC_ENCRYPT_DATA							✓
CKM_DES3_ECB_ENCRYPT_DATA							✓
CKM_DES3_CBC_ENCRYPT_DATA							✓
CKM_AES_ECB_ENCRYPT_DATA							✓
CKM_AES_CBC_ENCRYPT_DATA							✓

3473 2.17.1 Definitions

3474 Mechanisms:

```

3475     CKM_DES_ECB_ENCRYPT_DATA
3476     CKM_DES_CBC_ENCRYPT_DATA
3477     CKM_DES3_ECB_ENCRYPT_DATA
3478     CKM_DES3_CBC_ENCRYPT_DATA
3479     CKM_AES_ECB_ENCRYPT_DATA
3480     CKM_AES_CBC_ENCRYPT_DATA

3481
3482     typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
3483         CK_BYTE           iv[8];
3484         CK_BYTE_PTR      pData;
3485         CK ULONG          length;
3486     } CK_DES_CBC_ENCRYPT_DATA_PARAMS;
3487
3488     typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
3489             CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;
3490

```

```

3491     typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
3492         CK_BYTE           iv[16];
3493         CK_BYTE_PTR      pData;
3494         CK ULONG          length;
3495     } CK_AES_CBC_ENCRYPT_DATA_PARAMS;
3496
3497     typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
3498     CK_AES_CBC_ENCRYPT_DATA_PARAMS_PTR;

```

3499 2.17.2 Mechanism Parameters

3500 Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 2.43.2

3501 *Table 94, Mechanism Parameters*

CKM_DES_ECB_ENCRYPT_DATA CKM_DES3_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 8 bytes long.
CKM_AES_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM DES CBC ENCRYPT DATA CKM DES3 CBC ENCRYPT DATA	Uses CK_DES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 8 byte IV value followed by the data. The data value part must be a multiple of 8 bytes long.
CKM_AES_CBC_ENCRYPT_DATA	Uses CK_AES_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

3502 2.17.3 Mechanism Description

3503 The mechanisms will function by performing the encryption over the data provided using the base key.
 3504 The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text
 3505 is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data.
 3506 The derived key shall be defined by the attribute template supplied but constrained by the length of cipher
 3507 text available for the key value and other normal PKCS11 derivation constraints.

3508 Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1
 3509 Key Derivation mechanism in section 2.20.5.

3510 If the data is too short to make the requested key then the mechanism returns
 3511 CKR_DATA_LEN_RANGE.

3512 2.18 Double and Triple-length DES

3513 *Table 95, Double and Triple-Length DES Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES2_KEY_GEN					✓		
CKM_DES3_KEY_GEN					✓		
CKM_DES3_ECB	✓					✓	
CKM_DES3_CBC	✓					✓	

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES3_CBC_PAD	✓					✓	
CKM_DES3_MAC_GENERAL		✓					
CKM_DES3_MAC		✓					

3514 2.18.1 Definitions

3515 This section defines the key type “CKK_DES2” and “CKK_DES3” for type CK_KEY_TYPE as used in the
 3516 CKA_KEY_TYPE attribute of key objects.

3517 Mechanisms:

3518 CKM_DES2_KEY_GEN
 3519 CKM_DES3_KEY_GEN
 3520 CKM_DES3_ECB
 3521 CKM_DES3_CBC
 3522 CKM_DES3_MAC
 3523 CKM_DES3_MAC_GENERAL
 3524 CKM_DES3_CBC_PAD

3525 2.18.2 DES2 secret key objects

3526 DES2 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES2**) hold double-length
 3527 DES keys. The following table defines the DES2 secret key object attributes, in addition to the common
 3528 attributes defined for this object class:

3529 *Table 96, DES2 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 16 bytes long)

3530 - Refer to [PKCS11-Base] table 11 for footnotes

3531 DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of
 3532 the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or
 3533 unwrap a DES2 key with incorrect parity will return an error.

3534 The following is a sample template for creating a double-length DES secret key object:

```
3535 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
3536 CK_KEY_TYPE keyType = CKK_DES2;
3537 CK_UTF8CHAR label[] = "A DES2 secret key object";
3538 CK_BYTE value[16] = {...};
3539 CK_BBOOL true = CK_TRUE;
3540 CK_ATTRIBUTE template[] = {
3541     {CKA_CLASS, &class, sizeof(class)},
3542     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3543     {CKA_TOKEN, &true, sizeof(true)},
3544     {CKA_LABEL, label, sizeof(label)-1},
3545     {CKA_ENCRYPT, &true, sizeof(true)},
3546     {CKA_VALUE, value, sizeof(value)} }
```

3547 } ;
3548
3549 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
3550 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
3551 the key type of the secret key object.

3552 2.18.3 DES3 secret key objects

3553 DES3 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES3**) hold triple-length DES
3554 keys. The following table defines the DES3 secret key object attributes, in addition to the common
3555 attributes defined for this object class:

3556 *Table 97, DES3 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 24 bytes long)

3557 - Refer to [PKCS11-Base] table 11 for footnotes

3558 DES3 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of
3559 the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or
3560 unwrap a DES3 key with incorrect parity will return an error.

3561 The following is a sample template for creating a triple-length DES secret key object:

```
3562 CK_OBJECT_CLASS class = CKO_SECRET_KEY;  
3563 CK_KEY_TYPE keyType = CKK_DES3;  
3564 CK_UTF8CHAR label[] = "A DES3 secret key object";  
3565 CK_BYTE value[24] = {...};  
3566 CK_BBOOL true = CK_TRUE;  
3567 CK_ATTRIBUTE template[] = {  
3568     {CKA_CLASS, &class, sizeof(class)},  
3569     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
3570     {CKA_TOKEN, &true, sizeof(true)},  
3571     {CKA_LABEL, label, sizeof(label)-1},  
3572     {CKA_ENCRYPT, &true, sizeof(true)},  
3573     {CKA_VALUE, value, sizeof(value)}  
3574 };
```

3575

3576 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
3577 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
3578 the key type of the secret key object.

3579 2.18.4 Double-length DES key generation

3580 The double-length DES key generation mechanism, denoted **CKM_DES2_KEY_GEN**, is a key
3581 generation mechanism for double-length DES keys. The DES keys making up a double-length DES key
3582 both have their parity bits set properly, as specified in FIPS PUB 46-3.

3583 It does not have a parameter.

3584 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
3585 key. Other attributes supported by the double-length DES key type (specifically, the flags indicating which
3586 functions the key supports) may be specified in the template for the key, or else are assigned default
3587 initial values.

3588 Double-length DES keys can be used with all the same mechanisms as triple-DES keys:

3589 **CKM_DES3_ECB**, **CKM_DES3_CBC**, **CKM_DES3_CBC_PAD**, **CKM_DES3_MAC_GENERAL**, and

3590 **CKM_DES3_MAC.** Triple-DES encryption with a double-length DES key is equivalent to encryption with
3591 a triple-length DES key with K1=K3 as specified in FIPS PUB 46-3.
3592 When double-length DES keys are generated, it is token-dependent whether or not it is possible for either
3593 of the component DES keys to be “weak” or “semi-weak” keys.

3594 **2.18.5 Triple-length DES Order of Operations**

3595 Triple-length DES encryptions are carried out as specified in FIPS PUB 46-3: encrypt, decrypt, encrypt.
3596 Decryptions are carried out with the opposite three steps: decrypt, encrypt, decrypt. The mathematical
3597 representations of the encrypt and decrypt operations are as follows:

$$3598 \quad \text{DES3-E}(\{K1, K2, K3\}, P) = E(K3, D(K2, E(K1, P))) \\ 3599 \quad \text{DES3-D}(\{K1, K2, K3\}, C) = D(K1, E(K2, D(K3, P)))$$

3600 **2.18.6 Triple-length DES in CBC Mode**

3601 Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer
3602 CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the
3603 CBC encrypt and decrypt operations are as follows:

$$3604 \quad \text{DES3-CBC-E}(\{K1, K2, K3\}, P) = E(K3, D(K2, E(K1, P + I))) \\ 3605 \quad \text{DES3-CBC-D}(\{K1, K2, K3\}, C) = D(K1, E(K2, D(K3, P))) + I$$

3606 The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the
3607 current input block. The addition operation is used is addition modulo-2 (XOR).

3608 **2.18.7 DES and Triple length DES in OFB Mode**

3609 Table 98, *DES and Triple Length DES in OFB Mode Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_DES_OFB64	✓						
CKM_DES_OFB8	✓						
CKM_DES_CFB64	✓						
CKM_DES_CFB8	✓						

3610
3611 Cipher DES has a output feedback mode, DES-OFB, denoted **CKM_DES_OFB8** and
3612 **CKM_DES_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.
3613 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
3614 the block size.
3615 Constraints on key types and the length of data are summarized in the following table:

3616 *Table 99, OFB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part
C_Decrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part

3617 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

2.18.8 DES and Triple length DES in CFB Mode

3619 Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**.
3620 It is a mechanism for single and multiple-part encryption and decryption with DES.

3621 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
3622 the block size.

3623 Constraints on key types and the length of data are summarized in the following table:

3624 *Table 100, CFB: Key And Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part
C_Decrypt	CKK_DES, CKK_DES2, CKK_DES3	any	same as input length	no final part

3625 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

2.19 Double and Triple-length DES CMAC

3626 *Table 101, Double and Triple-length DES CMAC Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_DES3_CMAC_GENERAL		✓					
CKM_DES3_CMAC		✓					

3628 ¹ SR = SignRecover, VR = VerifyRecover.

3629

3630 The following additional DES3 mechanisms have been added.

2.19.1 Definitions

3632 Mechanisms:

3633 CKM_DES3_CMAC_GENERAL

3634 CKM_DES3_CMAC

3635 **2.19.2 Mechanism parameters**

3636 CKM_DES3_CMAC_GENERAL uses the existing **CK_MAC_GENERAL_PARAMS** structure.
3637 CKM_DES3_CMAC does not use a mechanism parameter.

3638 **2.19.3 General-length DES3-MAC**

3639 General-length DES3-CMAC, denoted **CKM_DES3_CMAC_GENERAL**, is a mechanism for single- and
3640 multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

3641 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
3642 desired from the mechanism.

3643 The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in
3644 the MACing process.

3645 Constraints on key types and the length of data are summarized in the following table:

3646 *Table 102, General-length DES3-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_DES3 CKK_DES2	any	1-block size, as specified in parameters
C_Verify	CKK_DES3 CKK_DES2	any	1-block size, as specified in parameters

3647 Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits
3648 (which means using the entire block for DES). The MAC length must be specified before the
3649 communication starts, and must not be changed during the lifetime of the key. It is the caller's
3650 responsibility to follow these rules.

3651 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3652 are not used

3653 **2.19.4 DES3-CMAC**

3654 DES3-CMAC, denoted **CKM_DES3_CMAC**, is a special case of the general-length DES3-CMAC
3655 mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the
3656 DES3 block length is the minimum output length recommended by [NIST sp800-38b].

3657 Constraints on key types and the length of data are summarized in the following table:

3658 *Table 103, DES3-CMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_DES3 CKK_DES2	any	Block size (8 bytes)
C_Verify	CKK_DES3 CKK_DES2	any	Block size (8 bytes)

3659 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3660 are not used.

3661 **2.20 SHA-1**

3662 *Table 104, SHA-1 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA_1				✓			
CKM_SHA_1_HMAC_GENERAL		✓					
CKM_SHA_1_HMAC		✓					
CKM_SHA1_KEY_DERIVATION							✓
CKM_SHA_1_KEY_GEN					✓		

3663 2.20.1 Definitions

3664 This section defines the key type "CKK_SHA_1_HMAC" for type CK_KEY_TYPE as used in the
 3665 CKA_KEY_TYPE attribute of key objects.

3666 Mechanisms:

3667 CKM_SHA_1
 3668 CKM_SHA_1_HMAC
 3669 CKM_SHA_1_HMAC_GENERAL
 3670 CKM_SHA1_KEY_DERIVATION
 3671 CKM_SHA_1_KEY_GEN

3672

3673 2.20.2 SHA-1 digest

3674 The SHA-1 mechanism, denoted **CKM_SHA_1**, is a mechanism for message digesting, following the
 3675 Secure Hash Algorithm with a 160-bit message digest defined in FIPS PUB 180-2.

3676 It does not have a parameter.

3677 Constraints on the length of input and output data are summarized in the following table. For single-part
 3678 digesting, the data and the digest may begin at the same location in memory.

3679 *Table 105, SHA-1: Data Length*

Function	Input length	Digest length
C_Digest	any	20

3680 2.20.3 General-length SHA-1-HMAC

3681 The general-length SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC_GENERAL**, is a
 3682 mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash
 3683 function. The keys it uses are generic secret keys and CKK_SHA_1_HMAC.

3684 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 3685 output. This length should be in the range 1-20 (the output size of SHA-1 is 20 bytes). Signatures
 3686 (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

3687 Table 106, General-length SHA-1-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret CKK_SHA_1_ HMAC	any	1-20, depending on parameters
C_Verify	generic secret CKK_SHA_1_ HMAC	any	1-20, depending on parameters

3688 **2.20.4 SHA-1-HMAC**

3689 The SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC**, is a special case of the general-length
3690 SHA-1-HMAC mechanism in Section 2.20.3.

3691 It has no parameter, and always produces an output of length 20.

3692 **2.20.5 SHA-1 key derivation**

3693 SHA-1 key derivation, denoted **CKM_SHA1_KEY_DERIVATION**, is a mechanism which provides the
3694 capability of deriving a secret key by digesting the value of another secret key with SHA-1.

3695 The value of the base key is digested once, and the result is used to make the value of derived secret
3696 key.

- 3697 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
3698 generic secret key. Its length will be 20 bytes (the output size of SHA-1).
- 3699 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
3700 will be a generic secret key of the specified length.
- 3701 • If no length was provided in the template, but a key type is, then that key type must have a well-
3702 defined length. If it does, then the key produced by this mechanism will be of the type specified in the
3703 template. If it doesn't, an error will be returned.
- 3704 • If both a key type and a length are provided in the template, the length must be compatible with that
3705 key type. The key produced by this mechanism will be of the specified type and length.

3706 If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key will be set
3707 properly.

3708 If the requested type of key requires more than 20 bytes, such as DES3, an error is generated.

3709 This mechanism has the following rules about key sensitivity and extractability:

- 3710 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
3711 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
3712 default value.
- 3713 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
3714 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
3715 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
3716 **CKA_SENSITIVE** attribute.
- 3717 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
3718 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
3719 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
3720 value from its **CKA_EXTRACTABLE** attribute.

3721 **2.20.6 SHA-1 HMAC key generation**

3722 The SHA-1-HMAC key generation mechanism, denoted **CKM_SHA_1_KEY_GEN**, is a key generation
3723 mechanism for NIST's SHA-1-HMAC.

3724 It does not have a parameter.

3725 The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the
 3726 **CKA_VALUE_LEN** attribute of the template for the key.
 3727 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 3728 key. Other attributes supported by the SHA-1-HMAC key type (specifically, the flags indicating which
 3729 functions the key supports) may be specified in the template for the key, or else are assigned default
 3730 initial values.
 3731 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 3732 specify the supported range of **CKM_SHA_1_HMAC** key sizes, in bytes.

3733 **2.21 SHA-224**

3734 *Table 107, SHA-224 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA224				✓			
CKM_SHA224_HMAC		✓					
CKM_SHA224_HMAC_GENERAL		✓					
CKM_SHA224_RSA_PKCS		✓					
CKM_SHA224_RSA_PKCS_PSS		✓					
CKM_SHA224_KEY_DERIVATION							✓
CKM_SHA224_KEY_GEN					✓		

3735 **2.21.1 Definitions**

3736 This section defines the key type “CKK_SHA224_HMAC” for type CK_KEY_TYPE as used in the
 3737 CKA_KEY_TYPE attribute of key objects.

3738 Mechanisms:

3739 CKM_SHA224
 3740 CKM_SHA224_HMAC
 3741 CKM_SHA224_HMAC_GENERAL
 3742 CKM_SHA224_KEY_DERIVATION
 3743 CKM_SHA224_KEY_GEN

3744 **2.21.2 SHA-224 digest**

3745 The SHA-224 mechanism, denoted **CKM_SHA224**, is a mechanism for message digesting, following the
 3746 Secure Hash Algorithm with a 224-bit message digest defined in 0.

3747 It does not have a parameter.

3748 Constraints on the length of input and output data are summarized in the following table. For single-part
 3749 digesting, the data and the digest may begin at the same location in memory.

3750 Table 108, SHA-224: Data Length

Function	Input length	Digest length
C_Digest	any	28

3751 2.21.3 General-length SHA-224-HMAC

3752 The general-length SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC_GENERAL**, is the
3753 same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based
3754 on the SHA-224 hash function and length of the output should be in the range 1-28. The keys it uses are
3755 generic secret keys and CKK_SHA224_HMAC. FIPS-198 compliant tokens may require the key length to
3756 be at least 14 bytes; that is, half the size of the SHA-224 hash output.

3757 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
3758 output. This length should be in the range 1-28 (the output size of SHA-224 is 28 bytes). FIPS-198
3759 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).
3760 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC
3761 output.

3762 Table 109, General-length SHA-224-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret CKK_SHA224_HMAC	Any	1-28, depending on parameters
C_Verify	generic secret CKK_SHA224_HMAC	Any	1-28, depending on parameters

3763 2.21.4 SHA-224-HMAC

3764 The SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC**, is a special case of the general-length
3765 SHA-224-HMAC mechanism.

3766 It has no parameter, and always produces an output of length 28.

3767 2.21.5 SHA-224 key derivation

3768 SHA-224 key derivation, denoted **CKM_SHA224_KEY_DERIVATION**, is the same as the SHA-1 key
3769 derivation mechanism in Section 12.21.5 except that it uses the SHA-224 hash function and the relevant
3770 length is 28 bytes.

3771 2.21.6 SHA-224 HMAC key generation

3772 The SHA-224-HMAC key generation mechanism, denoted **CKM_SHA224_KEY_GEN**, is a key
3773 generation mechanism for NIST's SHA224-HMAC.

3774 It does not have a parameter.

3775 The mechanism generates SHA224-HMAC keys with a particular length in bytes, as specified in the
3776 **CKA_VALUE_LEN** attribute of the template for the key.

3777 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
3778 key. Other attributes supported by the SHA224-HMAC key type (specifically, the flags indicating which
3779 functions the key supports) may be specified in the template for the key, or else are assigned default
3780 initial values.

3781 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3782 specify the supported range of **CKM_SHA224_HMAC** key sizes, in bytes.

3783 **2.22 SHA-256**

3784 *Table 110, SHA-256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA256				✓			
CKM_SHA256_HMAC_GENERAL		✓					
CKM_SHA256_HMAC		✓					
CKM_SHA256_KEY_DERIVATION							✓
CKM_SHA256_KEY_GEN					✓		

3785 **2.22.1 Definitions**

3786 This section defines the key type “CKK_SHA256_HMAC” for type CK_KEY_TYPE as used in the
3787 CKA_KEY_TYPE attribute of key objects.

3788 Mechanisms:

3789 CKM_SHA256
3790 CKM_SHA256_HMAC
3791 CKM_SHA256_HMAC_GENERAL
3792 CKM_SHA256_KEY_DERIVATION
3793 CKM_SHA256_KEY_GEN

3794 **2.22.2 SHA-256 digest**

3795 The SHA-256 mechanism, denoted **CKM_SHA256**, is a mechanism for message digesting, following the
3796 Secure Hash Algorithm with a 256-bit message digest defined in FIPS PUB 180-2.

3797 It does not have a parameter.

3798 Constraints on the length of input and output data are summarized in the following table. For single-part
3799 digesting, the data and the digest may begin at the same location in memory.

3800 *Table 111, SHA-256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

3801 **2.22.3 General-length SHA-256-HMAC**

3802 The general-length SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC_GENERAL**, is the
3803 same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC
3804 construction based on the SHA-256 hash function and length of the output should be in the range 1-32.
3805 The keys it uses are generic secret keys and CKK_SHA256_HMAC. FIPS-198 compliant tokens may
3806 require the key length to be at least 16 bytes; that is, half the size of the SHA-256 hash output.

3807 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
3808 output. This length should be in the range 1-32 (the output size of SHA-256 is 32 bytes). FIPS-198
3809 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
3810 Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC
3811 output.

3812 Table 112, General-length SHA-256-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret, CKK_SHA256_HMAC	Any	1-32, depending on parameters

3813

2.22.4 SHA-256-HMAC

3814 The SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC**, is a special case of the general-length
 3815 SHA-256-HMAC mechanism in Section 2.22.3.

3816 It has no parameter, and always produces an output of length 32.

3817

2.22.5 SHA-256 key derivation

3818 SHA-256 key derivation, denoted CKM_SHA256_KEY_DERIVATION, is the same as the SHA-1 key
 3819 derivation mechanism in Section 2.20.5, except that it uses the SHA-256 hash function and the relevant
 3820 length is 32 bytes.

3821

2.22.6 SHA-256 HMAC key generation

3822 The SHA-256-HMAC key generation mechanism, denoted **CKM_SHA256_KEY_GEN**, is a key
 3823 generation mechanism for NIST's SHA256-HMAC.

3824 It does not have a parameter.

3825 The mechanism generates SHA256-HMAC keys with a particular length in bytes, as specified in the
 3826 **CKA_VALUE_LEN** attribute of the template for the key.

3827 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 3828 key. Other attributes supported by the SHA256-HMAC key type (specifically, the flags indicating which
 3829 functions the key supports) may be specified in the template for the key, or else are assigned default
 3830 initial values.

3831 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 3832 specify the supported range of **CKM_SHA256_HMAC** key sizes, in bytes.

3833

2.23 SHA-384

3834 Table 113, SHA-384 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA384				✓			
CKM_SHA384_HMAC_GENERAL		✓					
CKM_SHA384_HMAC		✓					
CKM_SHA384_KEY_DERIVATION							✓
CKM_SHA384_KEY_GEN					✓		

3835 **2.23.1 Definitions**

3836 This section defines the key type “CKK_SHA384_HMAC” for type CK_KEY_TYPE as used in the
3837 CKA_KEY_TYPE attribute of key objects.

3838 CKM_SHA384
3839 CKM_SHA384_HMAC
3840 CKM_SHA384_HMAC_GENERAL
3841 CKM_SHA384_KEY_DERIVATION
3842 CKM_SHA384_KEY_GEN

3843 **2.23.2 SHA-384 digest**

3844 The SHA-384 mechanism, denoted **CKM_SHA384**, is a mechanism for message digesting, following the
3845 Secure Hash Algorithm with a 384-bit message digest defined in FIPS PUB 180-2.

3846 It does not have a parameter.

3847 Constraints on the length of input and output data are summarized in the following table. For single-part
3848 digesting, the data and the digest may begin at the same location in memory.

3849 *Table 114, SHA-384: Data Length*

Function	Input length	Digest length
C_Digest	any	48

3850 **2.23.3 General-length SHA-384-HMAC**

3851 The general-length SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC_GENERAL**, is the
3852 same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC
3853 construction based on the SHA-384 hash function and length of the output should be in the range 1-48.

3854 The keys it uses are generic secret keys and CKK_SHA384_HMAC. FIPS-198 compliant tokens may
3855 require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.

3856 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
3857 output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198
3858 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).
3859 Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC
3860 output.

3861 *Table 115, General-length SHA-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA384_HMAC	Any	1-48, depending on parameters
C_Verify	generic secret, CKK_SHA384_HMAC	Any	1-48, depending on parameters

3862

3863 **2.23.4 SHA-384-HMAC**

3864 The SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC**, is a special case of the general-length
3865 SHA-384-HMAC mechanism.

3866 It has no parameter, and always produces an output of length 48.

3867 **2.23.5 SHA-384 key derivation**

3868 SHA-384 key derivation, denoted **CKM_SHA384_KEY_DERIVATION**, is the same as the SHA-1 key
3869 derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant
3870 length is 48 bytes.

3871 **2.23.6 SHA-384 HMAC key generation**

3872 The SHA-384-HMAC key generation mechanism, denoted **CKM_SHA384_KEY_GEN**, is a key
3873 generation mechanism for NIST's SHA384-HMAC.

3874 It does not have a parameter.

3875 The mechanism generates SHA384-HMAC keys with a particular length in bytes, as specified in the
3876 **CKA_VALUE_LEN** attribute of the template for the key.

3877 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
3878 key. Other attributes supported by the SHA384-HMAC key type (specifically, the flags indicating which
3879 functions the key supports) may be specified in the template for the key, or else are assigned default
3880 initial values.

3881 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3882 specify the supported range of **CKM_SHA384_HMAC** key sizes, in bytes.

3883 **2.24 SHA-512**

3884 *Table 116, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA512				✓			
CKM_SHA512_HMAC_GENERAL		✓					
CKM_SHA512_HMAC		✓					
CKM_SHA512_KEY_DERIVATION							✓
CKM_SHA512_KEY_GEN					✓		

3885 **2.24.1 Definitions**

3886 This section defines the key type "CKK_SHA512_HMAC" for type CK_KEY_TYPE as used in the
3887 CKA_KEY_TYPE attribute of key objects.

3888 Mechanisms:

3889 CKM_SHA512

3890 CKM_SHA512_HMAC

3891 CKM_SHA512_HMAC_GENERAL

3892 CKM_SHA512_KEY_DERIVATION

3893 CKM_SHA512_KEY_GEN

3894 **2.24.2 SHA-512 digest**

3895 The SHA-512 mechanism, denoted **CKM_SHA512**, is a mechanism for message digesting, following the
3896 Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

3897 It does not have a parameter.

3898 Constraints on the length of input and output data are summarized in the following table. For single-part
3899 digesting, the data and the digest may begin at the same location in memory.

3900 *Table 117, SHA-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

3901 **2.24.3 General-length SHA-512-HMAC**

3902 The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC_GENERAL**, is the
3903 same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC
3904 construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

3905 The keys it uses are generic secret keys and CKK_SHA512_HMAC. FIPS-198 compliant tokens may
3906 require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

3907 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
3908 output. This length should be in the range 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198
3909 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length).
3910 Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC
3911 output.

3912 *Table 118, General-length SHA-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA512_HMAC	Any	1-64, depending on parameters
C_Verify	generic secret, CKK_SHA512_HMAC	Any	1-64, depending on parameters

3913

3914 **2.24.4 SHA-512-HMAC**

3915 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length
3916 SHA-512-HMAC mechanism.

3917 It has no parameter, and always produces an output of length 64.

3918 **2.24.5 SHA-512 key derivation**

3919 SHA-512 key derivation, denoted **CKM_SHA512_KEY_DERIVATION**, is the same as the SHA-1 key
3920 derivation mechanism in Section 2.20.5, except that it uses the SHA-512 hash function and the relevant
3921 length is 64 bytes.

3922 **2.24.6 SHA-512 HMAC key generation**

3923 The SHA-512-HMAC key generation mechanism, denoted **CKM_SHA512_KEY_GEN**, is a key
3924 generation mechanism for NIST's SHA512-HMAC.

3925 It does not have a parameter.

3926 The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the
3927 **CKA_VALUE_LEN** attribute of the template for the key.

3928 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
3929 key. Other attributes supported by the SHA512-HMAC key type (specifically, the flags indicating which
3930 functions the key supports) may be specified in the template for the key, or else are assigned default
3931 initial values.

3932 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3933 specify the supported range of **CKM_SHA512_HMAC** key sizes, in bytes.

3934 **2.25 SHA-512/224**

3935 *Table 119, SHA-512/224 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_224				✓			
CKM_SHA512_224_HMAC_GENERAL		✓					
CKM_SHA512_224_HMAC		✓					
CKM_SHA512_224_KEY_DERIVATION							✓
CKM_SHA512_224_KEY_GEN					✓		

3936 **2.25.1 Definitions**

3937 This section defines the key type “CKK_SHA512_224_HMAC” for type CK_KEY_TYPE as used in the
3938 CKA_KEY_TYPE attribute of key objects.

3939 Mechanisms:

3940 CKM_SHA512_224
3941 CKM_SHA512_224_HMAC
3942 CKM_SHA512_224_HMAC_GENERAL
3943 CKM_SHA512_224_KEY_DERIVATION
3944 CKM_SHA512_224_KEY_GEN

3945 **2.25.2 SHA-512/224 digest**

3946 The SHA-512/224 mechanism, denoted **CKM_SHA512_224**, is a mechanism for message digesting,
3947 following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit
3948 message digest with a distinct initial hash value and truncated to 224 bits. **CKM_SHA512_224** is the
3949 same as **CKM_SHA512_T** with a parameter value of 224.

3950 It does not have a parameter.

3951 Constraints on the length of input and output data are summarized in the following table. For single-part
3952 digesting, the data and the digest may begin at the same location in memory.

3953 *Table 120, SHA-512/224: Data Length*

Function	Input length	Digest length
C_Digest	any	28

3954 **2.25.3 General-length SHA-512/224-HMAC**

3955 The general-length SHA-512/224-HMAC mechanism, denoted **CKM_SHA512_224_HMAC_GENERAL**,
3956 is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the
3957 HMAC construction based on the SHA-512/224 hash function and length of the output should be in the
3958 range 1-28. The keys it uses are generic secret keys and CKK_SHA512_224_HMAC. FIPS-198

3959 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-
3960 512/224 hash output.

3961 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
3962 output. This length should be in the range 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198
3963 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).
3964 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC
3965 output.

3966 *Table 121, General-length SHA-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA512_224_HMAC	Any	1-28, depending on parameters
C_Verify	generic secret, CKK_SHA512_224_HMAC	Any	1-28, depending on parameters

3967

2.25.4 SHA-512/224-HMAC

3969 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_224_HMAC**, is a special case of the general-
3970 length SHA-512/224-HMAC mechanism.

3971 It has no parameter, and always produces an output of length 28.

2.25.5 SHA-512/224 key derivation

3973 The SHA-512/224 key derivation, denoted **CKM_SHA512_224_KEY_DERIVATION**, is the same as the
3974 SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/224 hash function
3975 and the relevant length is 28 bytes.

2.25.6 SHA-512/224 HMAC key generation

3977 The SHA-512/224-HMAC key generation mechanism, denoted **CKM_SHA512_224_KEY_GEN**, is a key
3978 generation mechanism for NIST's SHA512/224-HMAC.

3979 It does not have a parameter.

3980 The mechanism generates SHA512/224-HMAC keys with a particular length in bytes, as specified in the
3981 **CKA_VALUE_LEN** attribute of the template for the key.

3982 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
3983 key. Other attributes supported by the SHA512/224-HMAC key type (specifically, the flags indicating
3984 which functions the key supports) may be specified in the template for the key, or else are assigned
3985 default initial values.

3986 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
3987 specify the supported range of **CKM_SHA512_224_HMAC** key sizes, in bytes.

2.26 SHA-512/256

3989 *Table 122, SHA-512/256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_256				✓			
CKM_SHA512_256_HMAC_GENERAL		✓					
CKM_SHA512_256_HMAC		✓					
CKM_SHA512_256_KEY_DERIVATION							✓
CKM_SHA512_256_KEY_GEN					✓		

3990 2.26.1 Definitions

3991 This section defines the key type "CKK_SHA512_256_HMAC" for type CK_KEY_TYPE as used in the
 3992 CKA_KEY_TYPE attribute of key objects.

3993 Mechanisms:

3994 CKM_SHA512_256
 3995 CKM_SHA512_256_HMAC
 3996 CKM_SHA512_256_HMAC_GENERAL
 3997 CKM_SHA512_256_KEY_DERIVATION
 3998 CKM_SHA512_256_KEY_GEN

3999 2.26.2 SHA-512/256 digest

4000 The SHA-512/256 mechanism, denoted **CKM_SHA512_256**, is a mechanism for message digesting,
 4001 following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit
 4002 message digest with a distinct initial hash value and truncated to 256 bits. **CKM_SHA512_256** is the
 4003 same as **CKM_SHA512_T** with a parameter value of 256.

4004 It does not have a parameter.

4005 Constraints on the length of input and output data are summarized in the following table. For single-part
 4006 digesting, the data and the digest may begin at the same location in memory.

4007 *Table 123, SHA-512/256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

4008 2.26.3 General-length SHA-512/256-HMAC

4009 The general-length SHA-512/256-HMAC mechanism, denoted **CKM_SHA512_256_HMAC_GENERAL**,
 4010 is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the
 4011 HMAC construction based on the SHA-512/256 hash function and length of the output should be in the
 4012 range 1-32. The keys it uses are generic secret keys and CKK_SHA512_256_HMAC. FIPS-198
 4013 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-
 4014 512/256 hash output.

4015 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
 4016 output. This length should be in the range 1-32 (the output size of SHA-512/256 is 32 bytes). FIPS-198
 4017 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).

4018 Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC
4019 output.

4020 *Table 124, General-length SHA-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret, CKK_SHA512_256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret, CKK_SHA512_256_HMAC	Any	1-32, depending on parameters

4021

2.26.4 SHA-512/256-HMAC

4023 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_256_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism.

4025 It has no parameter, and always produces an output of length 32.

2.26.5 SHA-512/256 key derivation

4027 The SHA-512/256 key derivation, denoted **CKM_SHA512_256_KEY_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/256 hash function and the relevant length is 32 bytes.

2.26.6 SHA-512/256 HMAC key generation

4031 The SHA-512/256-HMAC key generation mechanism, denoted **CKM_SHA512_256_KEY_GEN**, is a key generation mechanism for NIST's SHA512/256-HMAC.

4033 It does not have a parameter.

4034 The mechanism generates SHA512/256-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

4036 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA512/256-HMAC key type (specifically, the flags indicating which functions the key supports) may be specified in the template for the key, or else are assigned default initial values.

4040 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA512_256_HMAC** key sizes, in bytes.

2.27 SHA-512/t

4043 *Table 125, SHA-512 / t Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_T				✓			
CKM_SHA512_T_HMAC_GENERAL		✓					
CKM_SHA512_T_HMAC		✓					
CKM_SHA512_T_KEY_DERIVATION							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_T_KEY_GEN					✓		

4044 2.27.1 Definitions

4045 This section defines the key type "CKK_SHA512_T_HMAC" for type CK_KEY_TYPE as used in the
 4046 CKA_KEY_TYPE attribute of key objects.

4047 Mechanisms:

4048 CKM_SHA512_T
 4049 CKM_SHA512_T_HMAC
 4050 CKM_SHA512_T_HMAC_GENERAL
 4051 CKM_SHA512_T_KEY_DERIVATION
 4052
 4053 CKK_SHA512_T_KEY_GEN

4054 2.27.2 SHA-512/t digest

4055 The SHA-512/t mechanism, denoted **CKM_SHA512_T**, is a mechanism for message digesting, following
 4056 the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message
 4057 digest with a distinct initial hash value and truncated to t bits.

4058 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in
 4059 bytes of the desired output should be in the range of 0 – ⌈t/8⌉, where 0 < t < 512, and t > 384.

4060 Constraints on the length of input and output data are summarized in the following table. For single-part
 4061 digesting, the data and the digest may begin at the same location in memory.

4062 *Table 126, SHA-512/256: Data Length*

Function	Input length	Digest length
C_Digest	any	⌈t/8⌉, where 0 < t < 512, and t > 384

4063 2.27.3 General-length SHA-512/t-HMAC

4064 The general-length SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC_GENERAL**, is the
 4065 same as the general-length SHA-1-HMAC mechanism in Section 2.20.3, except that it uses the HMAC
 4066 construction based on the SHA-512/t hash function and length of the output should be in the range 0 – ⌈
 4067 t/8⌉, where 0 < t < 512, and t > 384.

4068 2.27.4 SHA-512/t-HMAC

4069 The SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC**, is a special case of the general-
 4070 length SHA-512/t-HMAC mechanism.

4071 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in
 4072 bytes of the desired output should be in the range of 0 – ⌈t/8⌉, where 0 < t < 512, and t > 384.

4073 **2.27.5 SHA-512/t key derivation**

4074 The SHA-512/t key derivation, denoted **CKM_SHA512_T_KEY_DERIVATION**, is the same as the SHA-
4075 512 key derivation mechanism in section 2.25.5, except that it uses the SHA-512/t hash function and the
4076 relevant length is $\lceil t/8 \rceil$ bytes, where $0 < t < 512$, and $t \geq 384$.

4077 **2.27.6 SHA-512/t HMAC key generation**

4078 The SHA-512/t-HMAC key generation mechanism, denoted **CKM_SHA512_T_KEY_GEN**, is a key
4079 generation mechanism for NIST's SHA512/t-HMAC.

4080 It does not have a parameter.

4081 The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the
4082 **CKA_VALUE_LEN** attribute of the template for the key.

4083 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4084 key. Other attributes supported by the SHA512/t-HMAC key type (specifically, the flags indicating which
4085 functions the key supports) may be specified in the template for the key, or else are assigned default
4086 initial values.

4087 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4088 specify the supported range of **CKM_SHA512_T_HMAC** key sizes, in bytes.

4089

4090 **2.28 SHA3-224**

4091 *Table 127, SHA-224 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_224				✓			
CKM_SHA3_224_HMAC		✓					
CKM_SHA3_224_HMAC_GENERAL		✓					
CKM_SHA3_224_KEY_DERIVATION							✓
CKM_SHA3_224_KEY_GEN					✓		

4092 **2.28.1 Definitions**

4093 Mechanisms:

4094 CKM_SHA3_224

4095 CKM_SHA3_224_HMAC

4096 CKM_SHA3_224_HMAC_GENERAL

4097 CKM_SHA3_224_KEY_DERIVATION

4098 CKM_SHA3_224_KEY_GEN

4099

4100 CKK_SHA3_224_HMAC

4101 **2.28.2 SHA3-224 digest**

4102 The SHA3-224 mechanism, denoted **CKM_SHA3_224**, is a mechanism for message digesting, following
4103 the Secure Hash 3 Algorithm with a 224-bit message digest defined in FIPS Pub 202.

4104 It does not have a parameter.

4105 Constraints on the length of input and output data are summarized in the following table. For single-part
4106 digesting, the data and the digest may begin at the same location in memory.

4107 *Table 128, SHA3-224: Data Length*

Function	Input length	Digest length
C_Digest	any	28

4108 **2.28.3 General-length SHA3-224-HMAC**

4109 The general-length SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC_GENERAL**, is the
4110 same as the general-length SHA-1-HMAC mechanism in section 2.20.4 except that it uses the HMAC
4111 construction based on the SHA3-224 hash function and length of the output should be in the range 1-28.
4112 The keys it uses are generic secret keys and CKK_SHA3_224_HMAC. FIPS-198 compliant tokens may
4113 require the key length to be at least 14 bytes; that is, half the size of the SHA3-224 hash output.

4114 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
4115 output. This length should be in the range 1-28 (the output size of SHA3-224 is 28 bytes). FIPS-198
4116 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).
4117 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 28-byte HMAC
4118 output.

4119 *Table 129, General-length SHA3-224-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_224_HMAC	Any	1-28, depending on parameters
C_Verify	generic secret or CKK_SHA3_224_HMAC	Any	1-28, depending on parameters

4120 **2.28.4 SHA3-224-HMAC**

4121 The SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC**, is a special case of the general-
4122 length SHA3-224-HMAC mechanism.

4123 It has no parameter, and always produces an output of length 28.

4124 **2.28.5 SHA3-224 key derivation**

4125 SHA-224 key derivation, denoted **CKM_SHA3_224_KEY_DERIVATION**, is the same as the SHA-1 key
4126 derivation mechanism in Section 2.20.5 except that it uses the SHA3-224 hash function and the relevant
4127 length is 28 bytes.

4128 **2.28.6 SHA3-224 HMAC key generation**

4129 The SHA3-224-HMAC key generation mechanism, denoted **CKM_SHA3_224_KEY_GEN**, is a key
4130 generation mechanism for NIST's SHA3-224-HMAC.

4131 It does not have a parameter.

4132 The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the
4133 **CKA_VALUE_LEN** attribute of the template for the key.

4134 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4135 key. Other attributes supported by the SHA3-224-HMAC key type (specifically, the flags indicating which

4136 functions the key supports) may be specified in the template for the key, or else are assigned default
 4137 initial values.
 4138 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 4139 specify the supported range of **CKM_SHA3_224_HMAC** key sizes, in bytes.

4140 2.29 SHA3-256

4141 *Table 130, SHA3-256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_256				✓			
CKM_SHA3_256_HMAC_GENERAL		✓					
CKM_SHA3_256_HMAC		✓					
CKM_SHA3_256_KEY_DERIVATION							✓
CKM_SHA3_256_KEY_GEN					✓		

4142 2.29.1 Definitions

4143 Mechanisms:
 4144 CKM_SHA3_256
 4145 CKM_SHA3_256_HMAC
 4146 CKM_SHA3_256_HMAC_GENERAL
 4147 CKM_SHA3_256_KEY_DERIVATION
 4148 CKM_SHA3_256_KEY_GEN
 4149
 4150 CKK_SHA3_256_HMAC

4151 2.29.2 SHA3-256 digest

4152 The SHA3-256 mechanism, denoted **CKM_SHA3_256**, is a mechanism for message digesting, following
 4153 the Secure Hash 3 Algorithm with a 256-bit message digest defined in FIPS PUB 202.
 4154 It does not have a parameter.
 4155 Constraints on the length of input and output data are summarized in the following table. For single-part
 4156 digesting, the data and the digest may begin at the same location in memory.
 4157 *Table 131, SHA3-256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

4158 2.29.3 General-length SHA3-256-HMAC

4159 The general-length SHA3-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC_GENERAL**, is the
 4160 same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC
 4161 construction based on the SHA3-256 hash function and length of the output should be in the range 1-32.
 4162 The keys it uses are generic secret keys and CKK_SHA3_256_HMAC. FIPS-198 compliant tokens may
 4163 require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.

4164 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
 4165 output. This length should be in the range 1-32 (the output size of SHA3-256 is 32 bytes). FIPS-198
 4166 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
 4167 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC
 4168 output.

4169 *Table 132, General-length SHA3-256-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret or CKK_SHA3_256_HMAC	Any	1-32, depending on parameters

4170 2.29.4 SHA3-256-HMAC

4171 The SHA-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC**, is a special case of the general-
 4172 length SHA-256-HMAC mechanism in Section 2.22.3.

4173 It has no parameter, and always produces an output of length 32.

4174 2.29.5 SHA3-256 key derivation

4175 SHA-256 key derivation, denoted CKM_SHA3_256_KEY_DERIVATION, is the same as the SHA-1 key
 4176 derivation mechanism in Section 2.20.5, except that it uses the SHA3-256 hash function and the relevant
 4177 length is 32 bytes.

4178 2.29.6 SHA3-256 HMAC key generation

4179 The SHA3-256-HMAC key generation mechanism, denoted **CKM_SHA3_256_KEY_GEN**, is a key
 4180 generation mechanism for NIST's SHA3-256-HMAC.

4181 It does not have a parameter.

4182 The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the
 4183 **CKA_VALUE_LEN** attribute of the template for the key.

4184 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 4185 key. Other attributes supported by the SHA3-256-HMAC key type (specifically, the flags indicating which
 4186 functions the key supports) may be specified in the template for the key, or else are assigned default
 4187 initial values.

4188 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 4189 specify the supported range of **CKM_SHA3_256_HMAC** key sizes, in bytes.

4190

4191 2.30 SHA3-384

4192 *Table 133, SHA3-384 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verif y	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_384				✓			
CKM_SHA3_384_HMAC_GENERAL		✓					
CKM_SHA3_384_HMAC		✓					

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verif y	SR & VR ¹	Diges t	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_384_KEY_DERIVATION							✓
CKM_SHA3_384_KEY_GEN				✓			

4193 2.30.1 Definitions

4194 CKM_SHA3_384
 4195 CKM_SHA3_384_HMAC
 4196 CKM_SHA3_384_HMAC_GENERAL
 4197 CKM_SHA3_384_KEY_DERIVATION
 4198 CKM_SHA3_384_KEY_GEN
 4199
 4200 CKK_SHA3_384_HMAC

4201 2.30.2 SHA3-384 digest

4202 The SHA3-384 mechanism, denoted **CKM_SHA3_384**, is a mechanism for message digesting, following
 4203 the Secure Hash 3 Algorithm with a 384-bit message digest defined in FIPS PUB 202.

4204 It does not have a parameter.

4205 Constraints on the length of input and output data are summarized in the following table. For single-part
 4206 digesting, the data and the digest may begin at the same location in memory.

4207 *Table 134, SHA3-384: Data Length*

Function	Input length	Digest length
C_Digest	any	48

4208 2.30.3 General-length SHA3-384-HMAC

4209 The general-length SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC construction based on the SHA-384 hash function and length of the output should be in the range 1-48. The keys it uses are generic secret keys and CKK_SHA3_384_HMAC. FIPS-198 compliant tokens may require the key length to be at least 24 bytes; that is, half the size of the SHA3-384 hash output.

4214

4215 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-48 (the output size of SHA3-384 is 48 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

4220 *Table 135, General-length SHA3-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_384_HMAC	Any	1-48, depending on parameters
C_Verify	generic secret or CKK_SHA3_384_HMAC	Any	1-48, depending on parameters

4221

4222 **2.30.4 SHA3-384-HMAC**

4223 The SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC**, is a special case of the general-
4224 length SHA3-384-HMAC mechanism.

4225 It has no parameter, and always produces an output of length 48.

4226 **2.30.5 SHA3-384 key derivation**

4227 SHA3-384 key derivation, denoted **CKM_SHA3_384_KEY_DERIVATION**, is the same as the SHA-1 key
4228 derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the relevant
4229 length is 48 bytes.

4230 **2.30.6 SHA3-384 HMAC key generation**

4231 The SHA3-384-HMAC key generation mechanism, denoted **CKM_SHA3_384_KEY_GEN**, is a key
4232 generation mechanism for NIST's SHA3-384-HMAC.

4233 It does not have a parameter.

4234 The mechanism generates SHA3-384-HMAC keys with a particular length in bytes, as specified in the
4235 **CKA_VALUE_LEN** attribute of the template for the key.

4236 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4237 key. Other attributes supported by the SHA3-384-HMAC key type (specifically, the flags indicating which
4238 functions the key supports) may be specified in the template for the key, or else are assigned default
4239 initial values.

4240 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4241 specify the supported range of **CKM_SHA3_384_HMAC** key sizes, in bytes.

4242 **2.31 SHA3-512**

4243 *Table 136, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_512				✓			
CKM_SHA3_512_HMAC_GENERAL		✓					
CKM_SHA3_512_HMAC		✓					
CKM_SHA3_512_KEY_DERIVATION							✓
CKM_SHA3_512_KEY_GEN				✓			

4244 **2.31.1 Definitions**

4245 CKM_SHA3_512
4246 CKM_SHA3_512_HMAC
4247 CKM_SHA3_512_HMAC_GENERAL
4248 CKM_SHA3_512_KEY_DERIVATION
4249 CKM_SHA3_512_KEY_GEN
4250
4251 CKK_SHA3_512_HMAC

4252 **2.31.2 SHA3-512 digest**

4253 The SHA3-512 mechanism, denoted **CKM_SHA3_512**, is a mechanism for message digesting, following
4254 the Secure Hash 3 Algorithm with a 512-bit message digest defined in FIPS PUB 202.

4255 It does not have a parameter.

4256 Constraints on the length of input and output data are summarized in the following table. For single-part
4257 digesting, the data and the digest may begin at the same location in memory.

4258 *Table 137, SHA3-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

4259 **2.31.3 General-length SHA3-512-HMAC**

4260 The general-length SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC_GENERAL**, is the
4261 same as the general-length SHA-1-HMAC mechanism in Section 2.20.4, except that it uses the HMAC
4262 construction based on the SHA3-512 hash function and length of the output should be in the range 1-
4263 64. The keys it uses are generic secret keys and CKK_SHA3_512_HMAC. FIPS-198 compliant tokens
4264 may require the key length to be at least 32 bytes; that is, half the size of the SHA3-512 hash output.

4265

4266 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
4267 output. This length should be in the range 1-64 (the output size of SHA3-512 is 64 bytes). FIPS-198
4268 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length).
4269 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC
4270 output.

4271 *Table 138, General-length SHA3-512-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_SHA3_512-HMAC	Any	1-64, depending on parameters
C_Verify	generic secret or CKK_SHA3_512_HMAC	Any	1-64, depending on parameters

4272

4273 **2.31.4 SHA3-512-HMAC**

4274 The SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC**, is a special case of the general-
4275 length SHA3-512-HMAC mechanism.

4276 It has no parameter, and always produces an output of length 64.

4277 2.31.5 SHA3-512 key derivation

4278 SHA3-512 key derivation, denoted **CKM_SHA3_512_KEY_DERIVATION**, is the same as the SHA-1 key
4279 derivation mechanism in Section 2.20.5, except that it uses the SHA-512 hash function and the relevant
4280 length is 64 bytes.

4281 2.31.6 SHA3-512 HMAC key generation

4282 The SHA3-512-HMAC key generation mechanism, denoted **CKM_SHA3_512_KEY_GEN**, is a key
4283 generation mechanism for NIST's SHA3-512-HMAC.

4284 It does not have a parameter.

4285 The mechanism generates SHA3-512-HMAC keys with a particular length in bytes, as specified in the
4286 **CKA_VALUE_LEN** attribute of the template for the key.

4287 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4288 key. Other attributes supported by the SHA3-512-HMAC key type (specifically, the flags indicating which
4289 functions the key supports) may be specified in the template for the key, or else are assigned default
4290 initial values.

4291 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4292 specify the supported range of **CKM_SHA3_512_HMAC** key sizes, in bytes.

4293 2.32 SHAKE

4294 *Table 139, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHAKE_128_KEY_DERIVATION							✓
CKM_SHAKE_256_KEY_DERIVATION							✓

4295 2.32.1 Definitions

4296 CKM_SHAKE_128_KEY_DERIVATION

4297 CKM_SHAKE_256_KEY_DERIVATION

4298 2.32.2 SHAKE Key Derivation

4299 SHAKE-128 and SHAKE-256 key derivation, denoted **CKM_SHAKE_128_KEY_DERIVATION** and
4300 **CKM_SHAKE_256_KEY_DERIVATION**, implements the SHAKE expansion function defined in FIPS 202
4301 on the input key.

- 4302 • If no length or key type is provided in the template a **CKR_INVALID_TEMPLATE** error is generated.
- 4303 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
4304 shall be a generic secret key of the specified length.
- 4305 • If no length was provided in the template, but a key type is, then that key type must have a well-
4306 defined length. If it does, then the key produced by this mechanism shall be of the type specified in
4307 the template. If it doesn't, an error shall be returned.
- 4308 • If both a key type and a length are provided in the template, the length must be compatible with that
4309 key type. The key produced by this mechanism shall be of the specified type and length.

4310 If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key shall be set
4311 properly.

- 4312 This mechanism has the following rules about key sensitivity and extractability:
- 4313 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
4314 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
4315 default value.
- 4316 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
4317 shall as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
4318 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
4319 **CKA_SENSITIVE** attribute.
- 4320 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then
4321 the derived key shall, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
4322 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
4323 value from its **CKA_EXTRACTABLE** attribute.

4324 **2.33 Blake2b-160**

4325 *Table 140, Blake2b-160 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_160				✓			
CKM_BLAKE2B_160_HMAC		✓					
CKM_BLAKE2B_160_HMAC_GENERAL		✓					
CKM_BLAKE2B_160_KEY_DERIVE							✓
CKM_BLAKE2B_160_KEY_GEN					✓		

4326 **2.33.1 Definitions**

4327 Mechanisms:

- 4328 CKM_BLAKE2B_160
- 4329 CKM_BLAKE2B_160_HMAC
- 4330 CKM_BLAKE2B_160_HMAC_GENERAL
- 4331 CKM_BLAKE2B_160_KEY_DERIVE
- 4332 CKM_BLAKE2B_160_KEY_GEN
- 4333 CKK_BLAKE2B_160_HMAC

4334 **2.33.2 BLAKE2B-160 digest**

4335 The BLAKE2B-160 mechanism, denoted **CKM_BLAKE2B_160**, is a mechanism for message digesting,
4336 following the Blake2b Algorithm with a 160-bit message digest without a key as defined in [RFC 7693](#).

4337 It does not have a parameter.

4338 Constraints on the length of input and output data are summarized in the following table. For single-part
4339 digesting, the data and the digest may begin at the same location in memory.

4340 *Table 141, BLAKE2B-160: Data Length*

Function	Input length	Digest length
C_Digest	any	20

4341 **2.33.3 General-length BLAKE2B-160-HMAC**

4342 The general-length BLAKE2B-160-HMAC mechanism, denoted
4343 **CKM_BLAKE2B_160_HMAC_GENERAL**, is the keyed variant of BLAKE2B-160 and length of the output
4344 should be in the range 1-20. The keys it uses are generic secret keys and CKK_BLAKE2B_160_HMAC.
4345 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
4346 output. This length should be in the range 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures
4347 (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

4348 *Table 142, General-length BLAKE2B-160-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_160_H MAC	Any	1-20, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_160_H MAC	Any	1-20, depending on parameters

4349 **2.33.4 BLAKE2B-160-HMAC**

4350 The BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC**, is a special case of the
4351 general-length BLAKE2B-160-HMAC mechanism.
4352 It has no parameter, and always produces an output of length 20.

4353 **2.33.5 BLAKE2B-160 key derivation**

4354 BLAKE2B-160 key derivation, denoted **CKM_BLAKE2B_160_KEY_DERIVE**, is the same as the SHA-1
4355 key derivation mechanism in Section 2.20.5 except that it uses the BLAKE2B-160 hash function and the
4356 relevant length is 20 bytes.

4357 **2.33.6 BLAKE2B-160 HMAC key generation**

4358 The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM_BLAKE2B_160_KEY_GEN**, is a
4359 key generation mechanism for BLAKE2B-160-HMAC.

4360 It does not have a parameter.

4361 The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the
4362 **CKA_VALUE_LEN** attribute of the template for the key.

4363 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4364 key. Other attributes supported by the BLAKE2B-160-HMAC key type (specifically, the flags indicating
4365 which functions the key supports) may be specified in the template for the key, or else are assigned
4366 default initial values.

4367 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4368 specify the supported range of **CKM_BLAKE2B_160_HMAC** key sizes, in bytes.

4369 **2.34 BLAKE2B-256**

4370 *Table 143, BLAKE2B-256 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_256				✓			
CKM_BLAKE2B_256_HMAC_GENERAL		✓					
CKM_BLAKE2B_256_HMAC		✓					
CKM_BLAKE2B_256_KEY_DERIVE							✓
CKM_BLAKE2B_256_KEY_GEN					✓		

2.34.1 Definitions

Mechanisms:

CKM_BLAKE2B_256

CKM_BLAKE2B_256_HMAC

CKM_BLAKE2B_256_HMAC_GENERAL

CKM_BLAKE2B_256_KEY_DERIVE

CKM_BLAKE2B_256_KEY_GEN

CKK_BLAKE2B_256_HMAC

2.34.2 BLAKE2B-256 digest

The BLAKE2B-256 mechanism, denoted **CKM_BLAKE2B_256**, is a mechanism for message digesting, following the Blake2b Algorithm with a 256-bit message digest without a key as defined in RFC 7693.

It does not have a parameter.

Constraints on the length of input and output data are summarized in the following table. For single-part digesting, the data and the digest may begin at the same location in memory.

Table 144, BLAKE2B-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

2.34.3 General-length BLAKE2B-256-HMAC

The general-length BLAKE2B-256-HMAC mechanism, denoted

CKM_BLAKE2B_256_HMAC_GENERAL, is the keyed variant of Blake2b-256 and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_BLAKE2B_256_HMAC.

It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired output. This length should be in the range 1-32 (the output size of BLAKE2B-256 is 32 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.

4393 Table 145, General-length BLAKE2B-256-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_256_HM_AC	Any	1-32, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_256_HM_AC	Any	1-32, depending on parameters

4394

2.34.4 BLAKE2B-256-HMAC

4395 The BLAKE2B-256-HMAC mechanism, denoted **CKM_BLAKE2B_256_HMAC**, is a special case of the
4396 general-length BLAKE2B-256-HMAC mechanism in Section 2.22.3.

4397 It has no parameter, and always produces an output of length 32.

4398

2.34.5 BLAKE2B-256 key derivation

4399 BLAKE2B-256 key derivation, denoted **CKM_BLAKE2B_256_KEY_DERIVE**, is the same as the SHA-1
4400 key derivation mechanism in Section 2.20.5, except that it uses the BLAKE2B-256 hash function and the
4401 relevant length is 32 bytes.

4402

2.34.6 BLAKE2B-256 HMAC key generation

4403 The BLAKE2B-256-HMAC key generation mechanism, denoted **CKM_BLAKE2B_256_KEY_GEN**, is a
4404 key generation mechanism for BLAKE2B-256-HMAC.

4405 It does not have a parameter.

4406 The mechanism generates BLAKE2B-256-HMAC keys with a particular length in bytes, as specified in the
4407 **CKA_VALUE_LEN** attribute of the template for the key.

4408 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4409 key. Other attributes supported by the BLAKE2B-256-HMAC key type (specifically, the flags indicating
4410 which functions the key supports) may be specified in the template for the key, or else are assigned
4411 default initial values.

4412 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4413 specify the supported range of **CKM_BLAKE2B_256_HMAC** key sizes, in bytes.

4414

2.35 BLAKE2B-384

4415 Table 146, BLAKE2B-384 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_384				✓			
CKM_BLAKE2B_384_HMAC_GENERAL		✓					
CKM_BLAKE2B_384_HMAC		✓					
CKM_BLAKE2B_384_KEY_DERIVE							✓
CKM_BLAKE2B_384_KEY_GEN				✓			

4416 **2.35.1 Definitions**

4417 CKM_BLAKE2B_384
4418 CKM_BLAKE2B_384_HMAC
4419 CKM_BLAKE2B_384_HMAC_GENERAL
4420 CKM_BLAKE2B_384_KEY_DERIVE
4421 CKM_BLAKE2B_384_KEY_GEN
4422 CKK_BLAKE2B_384_HMAC

4423 **2.35.2 BLAKE2B-384 digest**

4424 The BLAKE2B-384 mechanism, denoted **CKM_BLAKE2B_384**, is a mechanism for message digesting,
4425 following the Blake2b Algorithm with a 384-bit message digest without a key as defined in RFC 7693.

4426 It does not have a parameter.

4427 Constraints on the length of input and output data are summarized in the following table. For single-part
4428 digesting, the data and the digest may begin at the same location in memory.

4429 *Table 147, BLAKE2B-384: Data Length*

Function	Input length	Digest length
C_Digest	any	48

4430 **2.35.3 General-length BLAKE2B-384-HMAC**

4431 The general-length BLAKE2B-384-HMAC mechanism, denoted
4432 **CKM_BLAKE2B_384_HMAC_GENERAL**, is the keyed variant of the Blake2b-384 hash function and
4433 length of the output should be in the range 1-48. The keys it uses are generic secret keys and
4434 CKK_BLAKE2B_384_HMAC.

4435

4436 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
4437 output. This length should be in the range 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures
4438 (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

4439 *Table 148, General-length BLAKE2B-384-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_384_H MAC	Any	1-48, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_384_H MAC	Any	1-48, depending on parameters

4440

4441 **2.35.4 BLAKE2B-384-HMAC**

4442 The BLAKE2B-384-HMAC mechanism, denoted **CKM_BLAKE2B_384_HMAC**, is a special case of the
4443 general-length BLAKE2B-384-HMAC mechanism.

4444 It has no parameter, and always produces an output of length 48.

4445 2.35.5 BLAKE2B-384 key derivation

4446 BLAKE2B-384 key derivation, denoted **CKM_BLAKE2B_384_KEY_DERIVE**, is the same as the SHA-1
4447 key derivation mechanism in Section 2.20.5, except that it uses the SHA-384 hash function and the
4448 relevant length is 48 bytes.

4449 2.35.6 BLAKE2B-384 HMAC key generation

4450 The BLAKE2B-384-HMAC key generation mechanism, denoted **CKM_BLAKE2B_384_KEY_GEN**, is a
4451 key generation mechanism for NIST's BLAKE2B-384-HMAC.

4452 It does not have a parameter.

4453 The mechanism generates BLAKE2B-384-HMAC keys with a particular length in bytes, as specified in the
4454 **CKA_VALUE_LEN** attribute of the template for the key.

4455 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4456 key. Other attributes supported by the BLAKE2B-384-HMAC key type (specifically, the flags indicating
4457 which functions the key supports) may be specified in the template for the key, or else are assigned
4458 default initial values.

4459 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4460 specify the supported range of **CKM_BLAKE2B_384_HMAC** key sizes, in bytes.

4461 2.36 BLAKE2B-512

4462 *Table 149, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_BLAKE2B_512				✓			
CKM_BLAKE2B_512_HMAC_GENERAL		✓					
CKM_BLAKE2B_512_HMAC		✓					
CKM_BLAKE2B_512_KEY_DERIVE							✓
CKM_BLAKE2B_512_KEY_GEN				✓			

4463 2.36.1 Definitions

4464 CKM_BLAKE2B_512

4465 CKM_BLAKE2B_512_HMAC

4466 CKM_BLAKE2B_512_HMAC_GENERAL

4467 CKM_BLAKE2B_512_KEY_DERIVE

4468 CKM_BLAKE2B_512_KEY_GEN

4469 CKK_BLAKE2B_512_HMAC

4470 2.36.2 BLAKE2B-512 digest

4471 The BLAKE2B-512 mechanism, denoted **CKM_BLAKE2B_512**, is a mechanism for message digesting,
4472 following the Blake2b Algorithm with a 512-bit message digest defined in RFC 7693.

4473 It does not have a parameter.

4474 Constraints on the length of input and output data are summarized in the following table. For single-part
4475 digesting, the data and the digest may begin at the same location in memory.

4476 *Table 150, BLAKE2B-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

4477 2.36.3 General-length BLAKE2B-512-HMAC

4478 The general-length BLAKE2B-512-HMAC mechanism, denoted
4479 **CKM_BLAKE2B_512_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-512 hash function and
4480 length of the output should be in the range 1-64. The keys it uses are generic secret keys and
4481 CKK_BLAKE2B_512_HMAC.

4482

4483 It has a parameter, a CK_MAC_GENERAL_PARAMS, which holds the length in bytes of the desired
4484 output. This length should be in the range 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures
4485 (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

4486 *Table 151, General-length BLAKE2B-512-HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_512- HMAC	Any	1-64, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_512_HM AC	Any	1-64, depending on parameters

4487

4488 2.36.4 BLAKE2B-512-HMAC

4489 The BLAKE2B-512-HMAC mechanism, denoted **CKM_BLAKE2B_512_HMAC**, is a special case of the
4490 general-length BLAKE2B-512-HMAC mechanism.

4491 It has no parameter, and always produces an output of length 64.

4492 2.36.5 BLAKE2B-512 key derivation

4493 BLAKE2B-512 key derivation, denoted **CKM_BLAKE2B_512_KEY_DERIVE**, is the same as the SHA-1
4494 key derivation mechanism in Section 2.20.5, except that it uses the Blake2b-512 hash function and the
4495 relevant length is 64 bytes.

4496 2.36.6 BLAKE2B-512 HMAC key generation

4497 The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM_BLAKE2B_512_KEY_GEN**, is a
4498 key generation mechanism for NIST's BLAKE2B-512-HMAC.

4499 It does not have a parameter.

4500 The mechanism generates BLAKE2B-512-HMAC keys with a particular length in bytes, as specified in the
4501 **CKA_VALUE_LEN** attribute of the template for the key.

4502 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4503 key. Other attributes supported by the BLAKE2B-512-HMAC key type (specifically, the flags indicating
4504 which functions the key supports) may be specified in the template for the key, or else are assigned
4505 default initial values.

4506 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
4507 specify the supported range of **CKM_BLAKE2B_512_HMAC** key sizes, in bytes.

4508

4509 2.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

4510 The mechanisms in this section are for generating keys and IVs for performing password-based
 4511 encryption. The method used to generate keys and IVs is specified in PKCS #5.

4512 *Table 152, PKCS 5 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_PBE_SHA1_DES3_EDE_CBC					✓		
CKM_PBE_SHA1_DES2_EDE_CBC					✓		
CKM_PBA_SHA1_WITH_SHA1_HMAC					✓		
CKM_PKCS5_PBKD2					✓		

4513 2.37.1 Definitions

4514 Mechanisms:

4515 CKM_PBE_SHA1_DES3_EDE_CBC
 4516 CKM_PBE_SHA1_DES2_EDE_CBC
 4517 CKM_PKCS5_PBKD2
 4518 CKM_PBA_SHA1_WITH_SHA1_HMAC

4519 2.37.2 Password-based encryption/authentication mechanism parameters

4520 ♦ CK_PBE_PARAMS; CK_PBE_PARAMS_PTR

4521 **CK_PBE_PARAMS** is a structure which provides all of the necessary information required by the
 4522 CKM_PBE mechanisms (see PKCS #5 and PKCS #12 for information on the PBE generation
 4523 mechanisms) and the CKM_PBA_SHA1_WITH_SHA1_HMAC mechanism. It is defined as follows:

```
4524     typedef struct CK_PBE_PARAMS {
4525         CK_BYTE_PTR          pInitVector;
4526         CK_UTF8CHAR_PTR      pPassword;
4527         CK ULONG             ulPasswordLen;
4528         CK_BYTE_PTR          pSalt;
4529         CK ULONG             ulSaltLen;
4530         CK ULONG             ulIteration;
4531     } CK_PBE_PARAMS;
```

4532
 4533 The fields of the structure have the following meanings:

4534 *pInitVector* pointer to the location that receives the 8-byte initialization vector
 4535 (IV), if an IV is required;

4536 *pPassword* points to the password to be used in the PBE key generation;

4537 *ulPasswordLen* length in bytes of the password information;

4538 *pSalt* *points to the salt to be used in the PBE key generation;*
 4539 *ulSaltLen* *length in bytes of the salt information;*
 4540 *ullIteration* *number of iterations required for the generation.*

4541 **CK_PBE_PARAMS_PTR** is a pointer to a **CK_PBE_PARAMS**.

4542 2.37.3 PKCS #5 PBKDF2 key generation mechanism parameters

4543 ♦ **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;**
 4544 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR**

4545 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE** is used to indicate the Pseudo-Random
 4546 Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

4547 `typedef CK ULONG CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;`

4548

4549 The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

4550 *Table 153, PKCS #5 PBKDF2 Key Generation: Pseudo-random functions*

PRF Identifier	Value	Parameter Type
CKP_PKCS5_PBKD2_HMAC_SHA1	0x00000001UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_GOSTR3411	0x00000002UL	This PRF uses GOST R34.11-94 hash to produce secret key value. <i>pPrfData</i> should point to DER-encoded OID, indicating GOSTR34.11-94 parameters. <i>ulPrfDataLen</i> holds encoded OID length in bytes. If <i>pPrfData</i> is set to <i>NULL_PTR</i> , then <i>id-GostR3411-94-CryptoProParamSet</i> parameters will be used (RFC 4357, 11.2), and <i>ulPrfDataLen</i> must be 0.
CKP_PKCS5_PBKD2_HMAC_SHA224	0x00000003UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA256	0x00000004UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA384	0x00000005UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA512	0x00000006UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
CKP_PKCS5_PBKD2_HMAC_SHA512_224	0x00000007UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.

CKP_PKCS5_PBKD2_HMAC_SHA512_256	0x00000008UL	No Parameter. <i>pPrfData</i> must be NULL and <i>ulPrfDataLen</i> must be zero.
---------------------------------	--------------	--

4551 CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR is a pointer to a
 4552 CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE.
 4553

- 4554 ◆ CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;
 4555 CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR

4556 CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE is used to indicate the source of the salt value when
 4557 deriving a key using PKCS #5 PBKDF2. It is defined as follows:

4558 `typedef CK ULONG CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;`

4559

4560 The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined
 4561 sources along with the corresponding data type for the *pSaltSourceData* field in the
 4562 CK_PKCS5_PBKD2_PARAM structure defined below.

4563 *Table 154, PKCS #5 PBKDF2 Key Generation: Salt sources*

Source Identifier	Value	Data Type
CKZ_SALT_SPECIFIED	0x00000001	Array of CK_BYTE containing the value of the salt value.

4564 CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR is a pointer to a
 4565 CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE.

- 4566 ◆ CK_PKCS5_PBKD2_PARAMS; CK_PKCS5_PBKD2_PARAMS_PTR

4567 CK_PKCS5_PBKD2_PARAMS is a structure that provides the parameters to the CKM_PKCS5_PBKD2
 4568 mechanism. The structure is defined as follows:

```
4569 typedef struct CK_PKCS5_PBKD2_PARAMS {
4570   CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE      saltSource;
4571   CK_VOID_PTR                         pSaltSourceData;
4572   CK ULONG                            ulSaltSourceDataLen;
4573   CK ULONG                            iterations;
4574   CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE prf;
4575   CK_VOID_PTR                         pPrfData;
4576   CK ULONG                            ulPrfDataLen;
4577   CK_UTF8CHAR_PTR                     pPassword;
4578   CK ULONG_PTR                        ulPasswordLen;
4579 } CK_PKCS5_PBKD2_PARAMS;
```

4580

4581 The fields of the structure have the following meanings:

4582 *saltSource* *source of the salt value*

4583 *pSaltSourceData* *data used as the input for the salt source*

4584 *ulSaltSourceDataLen* *length of the salt source input*

4585	<i>iterations</i>	<i>number of iterations to perform when generating each block of random data</i>
4586		
4587	<i>prf</i>	<i>pseudo-random function used to generate the key</i>
4588	<i>pPrfData</i>	<i>data used as the input for PRF in addition to the salt value</i>
4589	<i>ulPrfDataLen</i>	<i>length of the input data for the PRF</i>
4590	<i>pPassword</i>	<i>points to the password to be used in the PBE key generation</i>
4591	<i>ulPasswordLen</i>	<i>length in bytes of the password information</i>

4592 **CK_PKCS5_PBKD2_PARAMS_PTR** is a pointer to a **CK_PKCS5_PBKD2_PARAMS**.

4593 2.37.4 PKCS #5 PBKD2 key generation

4594 PKCS #5 PBKDF2 key generation, denoted **CKM_PKCS5_PBKD2**, is a mechanism used for generating
 4595 a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.
 4596 It has a parameter, a **CK_PKCS5_PBKD2_PARAMS** structure. The parameter specifies the salt value
 4597 source, pseudo-random function, and iteration count used to generate the new key.
 4598 Since this mechanism can be used to generate any type of secret key, new key templates must contain
 4599 the **CKA_KEY_TYPE** and **CKA_VALUE_LEN** attributes. If the key type has a fixed length the
 4600 **CKA_VALUE_LEN** attribute may be omitted.

4601 2.38 PKCS #12 password-based encryption/authentication 4602 mechanisms

4603 The mechanisms in this section are for generating keys and IVs for performing password-based
 4604 encryption or authentication. The method used to generate keys and IVs is based on a method that was
 4605 specified in PKCS #12.

4606 We specify here a general method for producing various types of pseudo-random bits from a password,
 4607 *p*; a string of salt bits, *s*; and an iteration count, *c*. The “type” of pseudo-random bits to be produced is
 4608 identified by an identification byte, *ID*, the meaning of which will be discussed later.

4609 Let *H* be a hash function built around a compression function $f: \mathbb{Z}_2^u \times \mathbb{Z}_2^v \rightarrow \mathbb{Z}_2^u$ (that is, *H* has a chaining
 4610 variable and output of length *u* bits, and the message input to the compression function of *H* is *v* bits).
 4611 For MD2 and MD5, *u*=128 and *v*=512; for SHA-1, *u*=160 and *v*=512.

4612 We assume here that *u* and *v* are both multiples of 8, as are the lengths in bits of the password and salt
 4613 strings and the number *n* of pseudo-random bits required. In addition, *u* and *v* are of course nonzero.

- 4614 1. Construct a string, *D* (the “diversifier”), by concatenating *v*/8 copies of *ID*.
- 4615 2. Concatenate copies of the salt together to create a string *S* of length $v - \lceil s/v \rceil$ bits (the final copy of the
 salt may be truncated to create *S*). Note that if the salt is the empty string, then so is *S*.
- 4616 3. Concatenate copies of the password together to create a string *P* of length $v - \lceil p/v \rceil$ bits (the final copy
 of the password may be truncated to create *P*). Note that if the password is the empty string, then so
 is *P*.
- 4617 4. Set *I*=*S||P* to be the concatenation of *S* and *P*.
- 4618 5. Set *j*= $\lceil n/u \rceil$.
- 4619 6. For *i*=1, 2, ..., *j*, do the following:
 - 4620 a. Set $A_i = H^c(D||I)$, the *c*th hash of *D||I*. That is, compute the hash of *D||I*; compute the hash of
 that hash; etc.; continue in this fashion until a total of *c* hashes have been computed, each on
 the result of the previous hash.

- 4626 b. Concatenate copies of A_i to create a string B of length v bits (the final copy of A_i may be
4627 truncated to create B).
4628 c. Treating I as a concatenation I_0, I_1, \dots, I_{k-1} of v -bit blocks, where $k=\lceil s/v \rceil + \lceil p/v \rceil$, modify I by
4629 setting $I_j = (I_j + B + 1) \bmod 2^v$ for each j . To perform this addition, treat each v -bit block as a
4630 binary number represented most-significant bit first.
4631 7. Concatenate A_1, A_2, \dots, A_k together to form a pseudo-random bit string, A .
4632 8. Use the first n bits of A as the output of this entire process.
4633 When the password-based encryption mechanisms presented in this section are used to generate a key
4634 and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To
4635 generate a key, the identifier byte ID is set to the value 1; to generate an IV, the identifier byte ID is set to
4636 the value 2.
4637 When the password based authentication mechanism presented in this section is used to generate a key
4638 from a password, salt, and an iteration count, the above algorithm is used. The identifier byte ID is set to
4639 the value 3.

4640 2.38.1 SHA-1-PBE for 3-key triple-DES-CBC

4641 SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES3_EDE_CBC**, is a mechanism
4642 used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the
4643 SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described
4644 above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-
4645 key triple-DES key with proper parity bits is obtained.

4646 It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the
4647 key generation process and the location of the application-supplied buffer which will receive the 8-byte IV
4648 generated by the mechanism.

4649 The key and IV produced by this mechanism will typically be used for performing password-based
4650 encryption.

4651 2.38.2 SHA-1-PBE for 2-key triple-DES-CBC

4652 SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES2_EDE_CBC**, is a mechanism
4653 used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the
4654 SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described
4655 above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-
4656 key triple-DES key with proper parity bits is obtained.

4657 It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the
4658 key generation process and the location of the application-supplied buffer which will receive the 8-byte IV
4659 generated by the mechanism.

4660 The key and IV produced by this mechanism will typically be used for performing password-based
4661 encryption.

4662 2.38.3 SHA-1-PBA for SHA-1-HMAC

4663 SHA-1-PBA for SHA-1-HMAC, denoted **CKM_PBA_SHA1_WITH_SHA1_HMAC**, is a mechanism used
4664 for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest
4665 algorithm and an iteration count. The method used to generate the key is described above.

4666 It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the
4667 key generation process. The parameter also has a field to hold the location of an application-supplied
4668 buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since
4669 authentication with SHA-1-HMAC does not require an IV.

4670 The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform
4671 password-based authentication (not *password-based encryption*). At the time of this writing, this is
4672 primarily done to ensure the integrity of a PKCS #12 PDU.

4673 2.39 SSL

4674 Table 155,SSL Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen - Key/Key Pair	Wrap & Unwrap	Derive
CKM_SSL3_PRE_MASTER_KEY_GEN					✓		
CKM_SSL3_MASTER_KEY_DERIVE							✓
CKM_SSL3_MASTER_KEY_DERIVE_DH							✓
CKM_SSL3_KEY_AND_MAC_DERIVE							✓
CKM_SSL3_MD5_MAC		✓					
CKM_SSL3_SHA1_MAC		✓					

4675 2.39.1 Definitions

4676 Mechanisms:

4677 CKM_SSL3_PRE_MASTER_KEY_GEN
4678 CKM_SSL3_MASTER_KEY_DERIVE
4679 CKM_SSL3_KEY_AND_MAC_DERIVE
4680 CKM_SSL3_MASTER_KEY_DERIVE_DH
4681 CKM_SSL3_MD5_MAC
4682 CKM_SSL3_SHA1_MAC

4683 2.39.2 SSL mechanism parameters

4684 ◆ CK_SSL3_RANDOM_DATA

4685 CK_SSL3_RANDOM_DATA is a structure which provides information about the random data of a client
4686 and a server in an SSL context. This structure is used by both the CKM_SSL3_MASTER_KEY_DERIVE
4687 and the CKM_SSL3_KEY_AND_MAC_DERIVE mechanisms. It is defined as follows:

```
4688     typedef struct CK_SSL3_RANDOM_DATA {  
4689         CK_BYTE_PTR    pClientRandom;  
4690         CK ULONG      ulClientRandomLen;  
4691         CK_BYTE_PTR    pServerRandom;  
4692         CK ULONG      ulServerRandomLen;  
4693     } CK_SSL3_RANDOM_DATA;
```

4694

4695 The fields of the structure have the following meanings:

4696 pClientRandom pointer to the client's random data

4697 ulClientRandomLen length in bytes of the client's random data

4698 pServerRandom pointer to the server's random data

4699 *ulServerRandomLen* *length in bytes of the server's random data*

4700 ♦ **CK_SSL3_MASTER_KEY_DERIVE_PARAMS;**
 4701 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR**

4702 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** is a structure that provides the parameters to the
 4703 **CKM_SSL3_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
4704        typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {
4705            CK_SSL3_RANDOM_DATA RandomInfo;
4706            CK_VERSION_PTR pVersion;
4707        } CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

4708

4709 The fields of the structure have the following meanings:

4710 *RandomInfo* *client's and server's random data information.*

4711 *pVersion* *pointer to a **CK_VERSION** structure which receives the SSL protocol version information*

4713 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
 4714 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**.

4715 ♦ **CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR**

4716 **CK_SSL3_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization vectors
 4717 after performing a C_DeriveKey function with the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It
 4718 is defined as follows:

```
4719        typedef struct CK_SSL3_KEY_MAT_OUT {
4720            CK_OBJECT_HANDLE hClientMacSecret;
4721            CK_OBJECT_HANDLE hServerMacSecret;
4722            CK_OBJECT_HANDLE hClientKey;
4723            CK_OBJECT_HANDLE hServerKey;
4724            CK_BYTE_PTR pIVClient;
4725            CK_BYTE_PTR pIVServer;
4726        } CK_SSL3_KEY_MAT_OUT;
```

4727

4728 The fields of the structure have the following meanings:

4729 *hClientMacSecret* *key handle for the resulting Client MAC Secret key*

4730 *hServerMacSecret* *key handle for the resulting Server MAC Secret key*

4731 *hClientKey* *key handle for the resulting Client Secret key*

4732 *hServerKey* *key handle for the resulting Server Secret key*

4733 *pIVClient* *pointer to a location which receives the initialization vector (IV) created for the client (if any)*

4735 *pIVServer* *pointer to a location which receives the initialization vector (IV) created for the server (if any)*

4737 **CK_SSL3_KEY_MAT_OUT_PTR** is a pointer to a **CK_SSL3_KEY_MAT_OUT**.

4738 ◆ **CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR**

4739 **CK_SSL3_KEY_MAT_PARAMS** is a structure that provides the parameters to the
4740 **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
4741     typedef struct CK_SSL3_KEY_MAT_PARAMS {  
4742         CK ULONG          ulMacSizeInBits;  
4743         CK ULONG          ulKeySizeInBits;  
4744         CK ULONG          ulIVSizeInBits;  
4745         CK_BBOOL          bIsExport;  
4746         CK_SSL3_RANDOM_DATA RandomInfo;  
4747         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
4748     } CK_SSL3_KEY_MAT_PARAMS;
```

4749

4750 The fields of the structure have the following meanings:

4751 *ulMacSizeInBits* *the length (in bits) of the MACing keys agreed upon during the*
4752 *protocol handshake phase*

4753 *ulKeySizeInBits* *the length (in bits) of the secret keys agreed upon during the*
4754 *protocol handshake phase*

4755 *ulIVSizeInBits* *the length (in bits) of the IV agreed upon during the protocol*
4756 *handshake phase. If no IV is required, the length should be set to 0*

4757 *bIsExport* *a Boolean value which indicates whether the keys have to be*
4758 *derived for an export version of the protocol*

4759 *RandomInfo* *client's and server's random data information.*

4760 *pReturnedKeyMaterial* *points to a CK_SSL3_KEY_MAT_OUT structures which receives*
4761 *the handles for the keys generated and the IVs*

4762 **CK_SSL3_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_SSL3_KEY_MAT_PARAMS**.

4763 **2.39.3 Pre-master key generation**

4764 Pre-master key generation in SSL 3.0, denoted **CKM_SSL3_PRE_MASTER_KEY_GEN**, is a mechanism
4765 which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL
4766 version 3.0 for RSA-like cipher suites.

4767 It has one parameter, a **CK_VERSION** structure, which provides the client's SSL version number.

4768 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4769 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
4770 be specified in the template, or else are assigned default values.

4771 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
4772 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
4773 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
4774 specify any of them.

4775 For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure
4776 both indicate 48 bytes.

4777 2.39.4 Master key derivation

4778 Master key derivation in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE**, is a mechanism used
4779 to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce
4780 the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the
4781 value of the client version, which is built into the "pre_master" key as well as a handle to the derived
4782 "master_secret" key.

4783 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
4784 passing of random data to the token as well as the returning of the protocol version number which is part
4785 of the pre-master key. This structure is defined in Section 2.39.

4786 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4787 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
4788 be specified in the template; otherwise they are assigned default values.

4789 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
4790 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
4791 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
4792 specify any of them.

4793 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
4795 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
4796 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
4798 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
4799 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
4800 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
4802 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
4803 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
4804 value from its **CKA_EXTRACTABLE** attribute.

4805 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
4806 both indicate 48 bytes.

4807 Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**
4808 structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns,
4809 this structure will hold the SSL version associated with the supplied pre_master key.

4810 Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an
4811 embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher
4812 suites.

4813 2.39.5 Master key derivation for Diffie-Hellman

4814 Master key derivation for Diffie-Hellman in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE_DH**,
4815 is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic
4816 secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master"
4817 key.

4818 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
4819 passing of random data to the token. This structure is defined in Section 2.39. The *pVersion* field of the
4820 structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key
4821 as it is for RSA-like cipher suites.

4822 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
4823 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
4824 be specified in the template, or else are assigned default values.

4825 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
4826 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
4827 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
4828 specify any of them.

4829 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
4831 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
4832 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
4834 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
4835 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
4836 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
4838 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
4839 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
4840 value from its **CKA_EXTRACTABLE** attribute.

4841 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
4842 both indicate 48 bytes.

4843 Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte
4844 "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but
4845 excludes the RSA cipher suites.

4846 2.39.6 Key and MAC derivation

4847 Key, MAC and IV derivation in SSL 3.0, denoted **CKM_SSL3_KEY_AND_MAC_DERIVE**, is a
4848 mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the
4849 "master_secret" key and random data. This mechanism returns the key handles for the keys generated in
4850 the process, as well as the IVs created.

4851 It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random
4852 data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a
4853 structure which receives the handles and IVs which were generated. This structure is defined in Section
4854 2.39.

4855 This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs
4856 are requested by the caller) back to the caller. The keys are all given an object class of
4857 **CKO_SECRET_KEY**.

4858 The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") are always given a
4859 type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing, verification, and derivation
4860 operations.

4861 The other two keys ("client_write_key" and "server_write_key") are typed according to information found
4862 in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are
4863 flagged as valid for encryption, decryption, and derivation operations.

4864 IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK_SSL3_KEY_MAT_PARAMS** field
4865 has a nonzero value. If they are generated, their length in bits will agree with the value in the
4866 *ulIVSizeInBits* field.

4867 All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
4868 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
4869 provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held
4870 by the base key.

4871 Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS**
4872 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four
4873 key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the
4874 newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's

4875 *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller).
 4876 Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.
 4877 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
 4878 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
 4879 successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns
 4880 all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the
 4881 **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
 4882 passed to **C_DeriveKey** is unnecessary, and should be a **NULL_PTR**.
 4883 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the
 4884 token.

4885 2.39.7 MD5 MACing in SSL 3.0

4886 MD5 MACing in SSL3.0, denoted **CKM_SSL3_MD5_MAC**, is a mechanism for single- and multiple-part
 4887 signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This
 4888 technique is very similar to the HMAC technique.

4889 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
 4890 signatures produced by this mechanism.

4891 Constraints on key types and the length of input and output data are summarized in the following table:

4892 *Table 156, MD5 MACing in SSL 3.0: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

4893 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 4894 specify the supported range of generic secret key sizes, in bits.

4895 2.39.8 SHA-1 MACing in SSL 3.0

4896 SHA-1 MACing in SSL3.0, denoted **CKM_SSL3_SHA1_MAC**, is a mechanism for single- and multiple-
 4897 part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This
 4898 technique is very similar to the HMAC technique.

4899 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
 4900 signatures produced by this mechanism.

4901 Constraints on key types and the length of input and output data are summarized in the following table:

4902 *Table 157, SHA-1 MACing in SSL 3.0: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	4-8, depending on parameters
C_Verify	generic secret	any	4-8, depending on parameters

4903 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 4904 specify the supported range of generic secret key sizes, in bits.

4905 2.40 TLS 1.2 Mechanisms

4906 Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2
 4907 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS
 4908 PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2
 4909 mechanisms contains a new value in the parameters structure to specify the hash function.

4910 This section also specifies CKM_TLS12_MAC which should be used in place of **CKM_TLS_PRF** to
 4911 calculate the verify_data in the TLS "finished" message.
 4912 This section also specifies **CKM_TLS_KDF** that can be used in place of **CKM_TLS_PRF** to implement
 4913 key material exporters.
 4914

4915 *Table 158, TLS 1.2 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_TLS12_MASTER_KEY_DERIVE							✓
CKM_TLS12_MASTER_KEY_DERIVE_DH							✓
CKM_TLS12_KEY_AND_MAC_DERIVE							✓
CKM_TLS12_KEY_SAFE_DERIVE							✓
CKM_TLS_KDF							✓
CKM_TLS12_MAC		✓					
CKM_TLS12_KDF							✓

4916 2.40.1 Definitions

4917 Mechanisms:

4918 CKM_TLS12_MASTER_KEY_DERIVE
 4919 CKM_TLS12_MASTER_KEY_DERIVE_DH
 4920 CKM_TLS12_KEY_AND_MAC_DERIVE
 4921 CKM_TLS12_KEY_SAFE_DERIVE
 4922 CKM_TLS_KDF
 4923 CKM_TLS12_MAC
 4924 CKM_TLS12_KDF

4925 2.40.2 TLS 1.2 mechanism parameters

- 4926 ◆ **CK_TLS12_MASTER_KEY_DERIVE_PARAMS;**
 4927 **CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR**

4928 **CK_TLS12_MASTER_KEY_DERIVE_PARAMS** is a structure that provides the parameters to the
 4929 **CKM_TLS12_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
4930     typedef struct CK_TLS12_MASTER_KEY_DERIVE_PARAMS {
4931         CK_SSL3_RANDOM_DATA RandomInfo;
4932         CK_VERSION_PTR pVersion;
4933         CK_MECHANISM_TYPE prfHashMechanism;
4934     } CK_TLS12_MASTER_KEY_DERIVE_PARAMS;
```

4935
 4936 The fields of the structure have the following meanings:

4937 *RandomInfo* *client's and server's random data information.*

4938 *pVersion* pointer to a **CK_VERSION** structure which receives the SSL
4939 protocol version information

4940 *prfHashMechanism* base hash used in the underlying TLS1.2 PRF operation used to
4941 derive the master key.

4942

4943 **CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
4944 **CK_TLS12_MASTER_KEY_DERIVE_PARAMS**.

4945 ◆ **CK_TLS12_KEY_MAT_PARAMS; CK_TLS12_KEY_MAT_PARAMS_PTR**

4946 **CK_TLS12_KEY_MAT_PARAMS** is a structure that provides the parameters to the
4947 **CKM_TLS12_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
4948       typedef struct CK_TLS12_KEY_MAT_PARAMS {  
4949           CK ULONG ulMacSizeInBits;  
4950           CK ULONG ulKeySizeInBits;  
4951           CK ULONG ullIVSizeInBits;  
4952           CK_BBOOL bIsExport;  
4953           CK_SSL3_RANDOM_DATA RandomInfo;  
4954           CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
4955           CK_MECHANISM_TYPE prfHashMechanism;  
4956       } CK_TLS12_KEY_MAT_PARAMS;
```

4957

4958 The fields of the structure have the following meanings:

4959 *ulMacSizeInBits* the length (in bits) of the MACing keys agreed upon during the
4960 protocol handshake phase. If no MAC key is required, the length
4961 should be set to 0.

4962 *ulKeySizeInBits* the length (in bits) of the secret keys agreed upon during the
4963 protocol handshake phase

4964 *ullIVSizeInBits* the length (in bits) of the IV agreed upon during the protocol
4965 handshake phase. If no IV is required, the length should be set to 0

4966 *bIsExport* must be set to *CK_FALSE* because export cipher suites must not be
4967 used in TLS 1.1 and later.

4968 *RandomInfo* client's and server's random data information.

4969 *pReturnedKeyMaterial* points to a **CK_SSL3_KEY_MAT_OUT** structures which receives
4970 the handles for the keys generated and the IVs

4971 *prfHashMechanism* base hash used in the underlying TLS1.2 PRF operation used to
4972 derive the master key.

4973 **CK_TLS12_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_TLS12_KEY_MAT_PARAMS**.

4974 ◆ **CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR**

4975 **CK_TLS_KDF_PARAMS** is a structure that provides the parameters to the **CKM_TLS_KDF** mechanism.
4976 It is defined as follows:

```

4977     typedef struct CK_TLS_KDF_PARAMS {
4978         CK_MECHANISM_TYPE prfMechanism;
4979         CK_BYTE_PTR pLabel;
4980         CK ULONG ulLabelLength;
4981         CK_SSL3_RANDOM_DATA RandomInfo;
4982         CK_BYTE_PTR pContextData;
4983         CK ULONG ulContextDataLength;
4984     } CK_TLS_KDF_PARAMS;
4985
4986 The fields of the structure have the following meanings:
4987 prfMechanism the hash mechanism used in the TLS1.2 PRF construct or CKM_TLS_PRF to use with the
4988 TLS1.0 and 1.1 PRF construct.
4989 pLabel a pointer to the label for this key derivation
4990 ulLabelLength length of the label in bytes
4991 RandomInfo the random data for the key derivation
4992 pContextData a pointer to the context data for this key derivation. NULL_PTR if not present
4993 ulContextDataLength length of the context data in bytes. 0 if not present.
4994

```

4995 ◆ **CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR**

4996 **CK_TLS_MAC_PARAMS** is a structure that provides the parameters to the **CKM_TLS_MAC**
 4997 mechanism. It is defined as follows:

```

4998     typedef struct CK_TLS_MAC_PARAMS {
4999         CK_MECHANISM_TYPE prfMechanism;
5000         CK ULONG ulMacLength;
5001         CK ULONG ulServerOrClient;
5002     } CK_TLS_MAC_PARAMS;
5003

```

5004 The fields of the structure have the following meanings:
 5005 *prfMechanism* the hash mechanism used in the TLS12 PRF construct or
 5006 CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
 5007 *ulMacLength* the length of the MAC tag required or offered. Always 12 octets in TLS 1.0 and 1.1.
 5008 Generally 12 octets, but may be negotiated to a longer value in
 5009 TLS1.2.
 5010 *ulServerOrClient* 1 to use the label "server finished", 2 to use the label "client
 5011 finished". All other values are invalid.
 5012 **CK_TLS_MAC_PARAMS_PTR** is a pointer to a **CK_TLS_MAC_PARAMS**.
 5013

5014 ◆ **CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR**

5015 **CK_TLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_TLS_PRF**
5016 mechanism. It is defined as follows:

```
5017     typedef struct CK_TLS_PRF_PARAMS {  
5018         CK_BYTE_PTR          pSeed;  
5019         CK ULONG             ulSeedLen;  
5020         CK_BYTE_PTR          pLabel;  
5021         CK ULONG             ulLabelLen;  
5022         CK_BYTE_PTR          pOutput;  
5023         CK ULONG_PTR         pulOutputLen;  
5024     } CK_TLS_PRF_PARAMS;
```

5025
5026 The fields of the structure have the following meanings:

5027

pSeed

 pointer to the input seed

5028

ulSeedLen

 length in bytes of the input seed

5029

pLabel

 pointer to the identifying label

5030

ulLabelLen

 length in bytes of the identifying label

5031

pOutput

 pointer receiving the output of the operation

5032

pulOutputLen

 pointer to the length in bytes that the output to be created shall
5033 have, has to hold the desired length as input and will receive the
5034 calculated length as output

5035 CK_TLS_PRF_PARAMS_PTR is a pointer to a CK_TLS_PRF_PARAMS.

5036 **2.40.3 TLS MAC**

5037 The TLS MAC mechanism is used to generate integrity tags for the TLS "finished" message. It replaces
5038 the use of the **CKM_TLS_PRF** function for TLS1.0 and 1.1 and that mechanism is deprecated.

5039 **CKM_TLS_MAC** takes a parameter of CK_TLS_MAC_PARAMS. To use this mechanism with TLS1.0
5040 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note:
5041 Although **CKM_TLS_PRF** is deprecated as a mechanism for C_DeriveKey, the manifest value is retained
5042 for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.

5043 In TLS1.0 and 1.1 the "finished" message verify_data (i.e. the output signature from the MAC mechanism)
5044 is always 12 bytes. In TLS1.2 the "finished" message verify_data is a minimum of 12 bytes, defaults to 12
5045 bytes, but may be negotiated to longer length.

5046 *Table 159, General-length TLS MAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	>=12 bytes
C_Verify	generic secret	any	>=12 bytes

5047
5048 **2.40.4 Master key derivation**

5049 Master key derivation in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE**, is a mechanism used to
5050 derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the

5051 "master_secret" key used in the TLS protocol from the "pre_master" key. This mechanism returns the
5052 value of the client version, which is built into the "pre_master" key as well as a handle to the derived
5053 "master_secret" key.

5054 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
5055 passing of random data to the token as well as the returning of the protocol version number which is part
5056 of the pre-master key. This structure is defined in Section 2.39.

5057 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
5058 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
5059 be specified in the template, or else are assigned default values.

5060 The mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of
5061 **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS12_KDF** and
5062 **CKM_TLS12_MAC**.

5063 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
5064 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
5065 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
5066 specify any of them.

5067 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
5069 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
5070 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
5071 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
5072 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
5073 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
5075 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
5076 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
5077 value from its **CKA_EXTRACTABLE** attribute.

5078 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
5079 both indicate 48 bytes.

5080 Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**
5081 structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns,
5082 this structure will hold the SSL version associated with the supplied pre_master key.

5083 Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an
5084 embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher
5085 suites.

5087 2.40.5 Master key derivation for Diffie-Hellman

5088 Master key derivation for Diffie-Hellman in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE_DH**, is
5089 a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret
5090 key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key.

5091 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
5092 passing of random data to the token. This structure is defined in Section 2.39. The *pVersion* field of the
5093 structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key
5094 as it is for RSA-like cipher suites.

5095 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
5096 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
5097 be specified in the template, or else are assigned default values.

5098 The mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of
5099 **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS12_KDF** and
5100 **CKM_TLS12_MAC**.

5101 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
5102 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
5103 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
5104 specify any of them.

5105 This mechanism has the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
5106 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
5107 default value.
- If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
5109 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
5110 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
5111 **CKA_SENSITIVE** attribute.
- Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
5113 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
5114 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
5115 value from its **CKA_EXTRACTABLE** attribute.

5117 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
5118 both indicate 48 bytes.

5119 Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte
5120 "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but
5121 excludes the RSA cipher suites.

5122 **2.40.6 Key and MAC derivation**

5123 Key, MAC and IV derivation in TLS 1.0, denoted **CKM_TLS_KEY_AND_MAC_DERIVE**, is a mechanism
5124 used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the
5125 "master_secret" key and random data. This mechanism returns the key handles for the keys generated in
5126 the process, as well as the IVs created.

5127 It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random
5128 data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a
5129 structure which receives the handles and IVs which were generated. This structure is defined in Section
5130 2.39.

5131 This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs
5132 are requested by the caller) back to the caller. The keys are all given an object class of
5133 **CKO_SECRET_KEY**.

5134 The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") (if present) are
5135 always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing and verification.

5136 The other two keys ("client_write_key" and "server_write_key") are typed according to information found
5137 in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are
5138 flagged as valid for encryption, decryption, and derivation operations.

5139 For **CKM_TLS12_KEY_AND_MAC_DERIVE**, IVs will be generated and returned if the *ullVSizeInBits*
5140 field of the **CK_SSL3_KEY_MAT_PARAMS** field has a nonzero value. If they are generated, their length
5141 in bits will agree with the value in the *ullVSizeInBits* field.

5142

5143 Note Well: CKM_TLS12_KEY_AND_MAC_DERIVE produces both private (key) and public (IV)
5144 data. It is possible to "leak" private data by the simple expedient of decreasing the length of
5145 private data requested. E.g. Setting ulMacSizeInBits and ulKeySizeInBits to 0 (or other lengths
5146 less than the key size) will result in the private key data being placed in the destination
5147 designated for the IV's. Repeated calls with the same master key and same RandomInfo but with
5148 differing lengths for the private key material will result in different data being leaked.<

5149

5150 All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
5151 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
5152 provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held
5153 by the base key.

5154 Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS**
5155 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four
5156 key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the
5157 newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's
5158 *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller).
5159 Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

5160 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
5161 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
5162 successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns
5163 all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the
5164 **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
5165 passed to **C_DeriveKey** is unnecessary, and should be a **NONE_PTR**.

5166 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the
5167 token.

5168 **2.40.7 CKM_TLS12_KEY_SAFE_DERIVE**

5169 **CKM_TLS12_KEY_SAFE_DERIVE** is identical to **CKM_TLS12_KEY_AND_MAC_DERIVE** except that it
5170 shall never produce IV data, and the *ullvSizelnBits* field of **CK_TLS12_KEY_MAT_PARAMS** is ignored
5171 and treated as 0. All of the other conditions and behavior described for
5172 **CKM_TLS12_KEY_AND_MAC_DERIVE**, with the exception of the black box warning, apply to this
5173 mechanism.

5174 **CKM_TLS12_KEY_SAFE_DERIVE** is provided as a separate mechanism to allow a client to control the
5175 export of IV material (and possible leaking of key material) through the use of the
5176 **CKA_ALLOWED_MECHANISMS** key attribute.

5177 **2.40.8 Generic Key Derivation using the TLS PRF**

5178 **CKM_TLS_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
5179 function to produce additional key material for protocols that want to leverage the TLS key negotiation
5180 mechanism. **CKM_TLS_KDF** has a parameter of **CK_TLS_KDF_PARAMS**. If the protocol using this
5181 mechanism does not use context information, the *pContextData* field shall be set to **NONE_PTR** and the
5182 *ulContextDataLength* field shall be set to 0.

5183 To use this mechanism with TLS1.0 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in
5184 place of a hash mechanism. Note: Although **CKM_TLS_PRF** is deprecated as a mechanism for
5185 **C_DeriveKey**, the manifest value is retained for use with this mechanism to indicate the use of the
5186 TLS1.0/1.1 Pseudo-random function.

5187 This mechanism can be used to derive multiple keys (e.g. similar to
5188 **CKM_TLS12_KEY_AND_MAC_DERIVE**) by first deriving the key stream as a **CKK_GENERIC_SECRET**
5189 of the necessary length and doing subsequent derives against that derived key using the
5190 **CKM_EXTRACT_KEY_FROM_KEY** mechanism to split the key stream into the actual operational keys.

5191 The mechanism should not be used with the labels defined for use with TLS, but the token does not
5192 enforce this behavior.

5193 This mechanism has the following rules about key sensitivity and extractability:

- 5194 • If the original key has its **CKA_SENSITIVE** attribute set to **CK_TRUE**, so does the derived key. If not,
5195 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from the
5196 original key.

- 5197 • Similarly, if the original key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 5198 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
 5199 supplied template or from the original key.
- 5200 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the original
 5201 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 5202 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 5203 the original key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

5204 2.40.9 Generic Key Derivation using the TLS12 PRF

5205 **CKM_TLS12_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
 5206 function to produce additional key material for protocols that want to leverage the TLS key negotiation
 5207 mechanism. **CKM_TLS12_KDF** has a parameter of **CK_TLS_KDF_PARAMS**. If the protocol using this
 5208 mechanism does not use context information, the *pContextData* field shall be set to NULL_PTR and the
 5209 *ulContextDataLength* field shall be set to 0.

5210 To use this mechanism with TLS1.0 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in
 5211 place of a hash mechanism. Note: Although **CKM_TLS_PRF** is deprecated as a mechanism for
 5212 C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the
 5213 TLS1.0/1.1 Pseudo-random function.

5214 This mechanism can be used to derive multiple keys (e.g. similar to
 5215 **CKM_TLS12_KEY_AND_MAC_DERIVE**) by first deriving the key stream as a **CKK_GENERIC_SECRET**
 5216 of the necessary length and doing subsequent derives against that derived key stream using the
 5217 **CKM_EXTRACT_KEY_FROM_KEY** mechanism to split the key stream into the actual operational keys.

5218 The mechanism should not be used with the labels defined for use with TLS, but the token does not
 5219 enforce this behavior.

5220 This mechanism has the following rules about key sensitivity and extractability:

- 5221 • If the original key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
 5222 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from the
 5223 original key.
- 5224 • Similarly, if the original key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 5225 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied
 5226 template or from the original key.
- 5227 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the original
 5228 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 5229 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 5230 the original key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

5231 2.41 WTLS

5232 Details can be found in [WTLS].

5233 When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue
 5234 that there would be no need to have distinct handling of the client and server side of the handshake.
 5235 However, since in WTLS the server and client use different sequence numbers, there could be instances
 5236 (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and
 5237 server side differ, and hence this motivates the introduced split.

5238

5239 Table 160, WTLS Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen - Key / Key Pair	Wrap & Unwrap	Derive
CKM_WTLS_PRE_MASTER_KEY_GEN					✓		
CKM_WTLS_MASTER_KEY_DERIVE							✓
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC							✓
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE							✓
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE							✓
CKM_WTLS_PRF							✓

5240 2.41.1 Definitions

5241 Mechanisms:

5242 CKM_WTLS_PRE_MASTER_KEY_GEN
 5243 CKM_WTLS_MASTER_KEY_DERIVE
 5244 CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC
 5245 CKM_WTLS_PRF
 5246 CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
 5247 CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE

5248 2.41.2 WTLS mechanism parameters

5249 ◆ CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR

5250 CK_WTLS_RANDOM_DATA is a structure, which provides information about the random data of a client
 5251 and a server in a WTLS context. This structure is used by the CKM_WTLS_MASTER_KEY_DERIVE
 5252 mechanism. It is defined as follows:

```
5253     typedef struct CK_WTLS_RANDOM_DATA {
5254         CK_BYTE_PTR pClientRandom;
5255         CK_ULONG     ulClientRandomLen;
5256         CK_BYTE_PTR pServerRandom;
5257         CK_ULONG     ulServerRandomLen;
5258     } CK_WTLS_RANDOM_DATA;
```

5260 The fields of the structure have the following meanings:

5261 pClientRandom pointer to the client's random data

5262 pClientRandomLen length in bytes of the client's random data

5263 pServerRaandom pointer to the server's random data

5264 *ulServerRandomLen* length in bytes of the server's random data
5265 **CK_WTLS_RANDOM_DATA_PTR** is a pointer to a **CK_WTLS_RANDOM_DATA**.
5266 ◆ **CK_WTLS_MASTER_KEY_DERIVE_PARAMS;**
5267 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR**

5268 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** is a structure, which provides the parameters to the
5269 **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
5270       typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {  
5271         CK_MECHANISM_TYPE DigestMechanism;  
5272         CK_WTLS_RANDOM_DATA RandomInfo;  
5273         CK_BYTE_PTR pVersion;  
5274     } CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
```

5275
5276 The fields of the structure have the following meanings:

5277 *DigestMechanism* the mechanism type of the digest mechanism to be used (possible
5278 types can be found in [WTLS])

5279 *RandomInfo* Client's and server's random data information

5280 *pVersion* pointer to a **CK_BYTE** which receives the WTLS protocol version
5281 information

5282 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
5283 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS**.

5284 ◆ **CK_WTLS_PRF_PARAMS; CK_WTLS_PRF_PARAMS_PTR**

5285 **CK_WTLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_WTLS_PRF**
5286 mechanism. It is defined as follows:

```
5287       typedef struct CK_WTLS_PRF_PARAMS {  
5288         CK_MECHANISM_TYPE DigestMechanism;  
5289         CK_BYTE_PTR pSeed;  
5290         CK ULONG ulSeedLen;  
5291         CK_BYTE_PTR pLabel;  
5292         CK ULONG ulLabelLen;  
5293         CK_BYTE_PTR pOutput;  
5294         CK ULONG_PTR pulOutputLen;  
5295     } CK_WTLS_PRF_PARAMS;
```

5296
5297 The fields of the structure have the following meanings:

5298 *Digest Mechanism* the mechanism type of the digest mechanism to be used (possible
5299 types can be found in [WTLS])

5300 *pSeed* pointer to the input seed

5301 *ulSeedLen* length in bytes of the input seed

5302 *pLabel* pointer to the identifying label

5303 *ulLabelLen* *length in bytes of the identifying label*
5304 *pOutput* *pointer receiving the output of the operation*
5305 *pulOutputLen* *pointer to the length in bytes that the output to be created shall*
5306 *have, has to hold the desired length as input and will receive the*
5307 *calculated length as output*

5308 **CK_WTLS_PRF_PARAMS_PTR** is a pointer to a **CK_WTLS_PRF_PARAMS**.

5309 ◆ **CK_WTLS_KEY_MAT_OUT; CK_WTLS_KEY_MAT_OUT_PTR**

5310 **CK_WTLS_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization
5311 vectors after performing a C_DeriveKey function with the
5312 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** or with the
5313 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
5314        typedef struct CK_WTLS_KEY_MAT_OUT {  
5315            CK_OBJECT_HANDLE hMacSecret;  
5316            CK_OBJECT_HANDLE hKey;  
5317            CK_BYTE_PTR      pIV;  
5318        } CK_WTLS_KEY_MAT_OUT;
```

5319

5320 The fields of the structure have the following meanings:

5321	<i>hMacSecret</i>	<i>Key handle for the resulting MAC secret key</i>
5322	<i>hKey</i>	<i>Key handle for the resulting secret key</i>
5323	<i>pIV</i>	<i>Pointer to a location which receives the initialization vector (IV) created (if any)</i>

5325 **CK_WTLS_KEY_MAT_OUT_PTR** is a pointer to a **CK_WTLS_KEY_MAT_OUT**.

5326 ◆ **CK_WTLS_KEY_MAT_PARAMS; CK_WTLS_KEY_MAT_PARAMS_PTR**

5327 **CK_WTLS_KEY_MAT_PARAMS** is a structure that provides the parameters to the
5328 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** and the
5329 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
5330        typedef struct CK_WTLS_KEY_MAT_PARAMS {  
5331            CK_MECHANISM_TYPE     DigestMechanism;  
5332            CK ULONG            ulMacSizeInBits;  
5333            CK ULONG            ulKeySizeInBits;  
5334            CK ULONG            ulIVSizeInBits;  
5335            CK ULONG            ulSequenceNumber;  
5336            CK_BBOOL            bIsExport;  
5337            CK_WTLS_RANDOM_DATA RandomInfo;  
5338            CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
5339        } CK_WTLS_KEY_MAT_PARAMS;
```

5340

5341 The fields of the structure have the following meanings:

5342	<i>Digest Mechanism</i>	<i>the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])</i>
5343		
5344	<i>ulMaxSizeInBits</i>	<i>the length (in bits) of the MACing key agreed upon during the protocol handshake phase</i>
5345		
5346	<i>ulKeySizeInBits</i>	<i>the length (in bits) of the secret key agreed upon during the handshake phase</i>
5347		
5348	<i>ullIVSizeInBits</i>	<i>the length (in bits) of the IV agreed upon during the handshake phase. If no IV is required, the length should be set to 0.</i>
5349		
5350	<i>ulSequenceNumber</i>	<i>the current sequence number used for records sent by the client and server respectively</i>
5351		
5352	<i>bIsExport</i>	<i>a boolean value which indicates whether the keys have to be derived for an export version of the protocol. If this value is true (i.e., the keys are exportable) then ulKeySizeInBits is the length of the key in bits before expansion. The length of the key after expansion is determined by the information found in the template sent along with this mechanism during a C_DeriveKey function call (either the CKA_KEY_TYPE or the CKA_VALUE_LEN attribute).</i>
5353		
5354		
5355		
5356		
5357		
5358		
5359	<i>RandomInfo</i>	<i>client's and server's random data information</i>
5360	<i>pReturnedKeyMaterial</i>	<i>points to a CK_WTLS_KEY_MAT_OUT structure which receives the handles for the keys generated and the IV</i>
5361		
5362	CK_WTLS_KEY_MAT_PARAMS_PTR	is a pointer to a CK_WTLS_KEY_MAT_PARAMS.

5363 **2.41.3 Pre master secret key generation for RSA key exchange suite**

5364 Pre master secret key generation for the RSA key exchange suite in WTLS denoted
 5365 **CKM_WTLS_PRE_MASTER_KEY_GEN**, is a mechanism, which generates a variable length secret key.
 5366 It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This
 5367 mechanism returns a handle to the pre master secret key.
 5368 It has one parameter, a **CK_BYTE**, which provides the client's WTLS version.
 5369 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE** and **CKA_VALUE** attributes to the new
 5370 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
 5371 be specified in the template, or else are assigned default values.
 5372 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
 5373 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
 5374 attribute indicates the length of the pre master secret key.
 5375 For this mechanism, the **ulMinKeySize** field of the **CK_MECHANISM_INFO** structure shall indicate 20
 5376 bytes.

5377 **2.41.4 Master secret key derivation**

5378 Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used
 5379 to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master
 5380 secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client
 5381 version, which is built into the pre master secret key as well as a handle to the derived master secret key.
 5382 It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for passing
 5383 the mechanism type of the digest mechanism to be used as well as the passing of random data to the
 5384 token as well as the returning of the protocol version number which is part of the pre master secret key.

5385 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
5386 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
5387 be specified in the template, or else are assigned default values.
5388 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
5389 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
5390 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to
5391 specify any of them.
5392 This mechanism has the following rules about key sensitivity and extractability:
5393 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be
5394 specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default
5395 value.
5396 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key will
5397 as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the derived
5398 key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.
5399 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
5400 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE,
5401 then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its
5402 **CKA_EXTRACTABLE** attribute.
5403 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
5404 both indicate 20 bytes.
5405 Note that the **CK_BYTE** pointed to by the **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure's
5406 *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this byte will
5407 hold the WTLS version associated with the supplied pre master secret key.
5408 Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret
5409 key with an embedded version number. This includes the RSA key exchange suites, but excludes the
5410 Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

5411 **2.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve 5412 Cryptography**

5413 Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted
5414 **CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC**, is a mechanism used to derive a 20 byte generic
5415 secret key from variable length secret key. It is used to produce the master secret key used in WTLS from
5416 the pre master secret key. This mechanism returns a handle to the derived master secret key.
5417 It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
5418 passing of the mechanism type of the digest mechanism to be used as well as random data to the token.
5419 The *pVersion* field of the structure must be set to NULL_PTR since the version number is not embedded
5420 in the pre master secret key as it is for RSA-like key exchange suites.
5421 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
5422 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
5423 be specified in the template, or else are assigned default values.
5424 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
5425 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
5426 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to
5427 specify any of them.
5428 This mechanism has the following rules about key sensitivity and extractability:
5429 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be
5430 specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some default
5431 value.
5432 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key will
5433 as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the derived
5434 key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

5435 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
5436 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE,
5437 then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its
5438 **CKA_EXTRACTABLE** attribute.
5439 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
5440 both indicate 20 bytes.
5441 Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte
5442 pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic
5443 Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

5444 **2.41.6 WTLS PRF (pseudorandom function)**

5445 PRF (pseudo random function) in WTLS, denoted **CKM_WTLS_PRF**, is a mechanism used to produce a
5446 securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.
5447 It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which allows for passing the mechanism type
5448 of the digest mechanism to be used, the passing of the input seed and its length, the passing of an
5449 identifying label and its length and the passing of the length of the output to the token and for receiving
5450 the output.
5451 This mechanism produces securely generated pseudo-random output of the length specified in the
5452 parameter.
5453 This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template
5454 sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a
5455 NULL_PTR. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result
5456 of a successful completion. However, since the **CKM_WTLS_PRF** mechanism returns the requested
5457 number of output bytes in the **CK_WTLS_PRF_PARAMS** structure specified as the mechanism
5458 parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.
5459 If a call to **C_DeriveKey** with this mechanism fails, then no output will be generated.

5460 **2.41.7 Server Key and MAC derivation**

5461 Server key, MAC and IV derivation in WTLS, denoted
5462 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate
5463 cryptographic keying material used by a cipher suite from the master secret key and random data. This
5464 mechanism returns the key handles for the keys generated in the process, as well as the IV created.
5465 It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the
5466 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic
5467 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which
5468 were generated.
5469 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested
5470 by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.
5471 The MACing key (server write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is
5472 flagged as valid for signing, verification and derivation operations.
5473 The other key (server write key) is typed according to information found in the template sent along with
5474 this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption,
5475 decryption, and derivation operations.
5476 An IV (server write IV) will be generated and returned if the *ulIVSizeInBits* field of the
5477 **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree
5478 with the value in the *ulIVSizeInBits* field.
5479 Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
5480 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
5481 provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by
5482 the base key.

5483 Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS**
5484 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key
5485 handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-
5486 created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will
5487 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a
5488 buffer with sufficient space to hold any IV that will be returned.
5489 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
5490 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
5491 successful completion. However, since the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**
5492 mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the
5493 **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
5494 passed to **C_DeriveKey** is unnecessary, and should be a **NONE_PTR**.
5495 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

5496 2.41.8 Client key and MAC derivation

5497 Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**,
5498 is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from
5499 the master secret key and random data. This mechanism returns the key handles for the keys generated
5500 in the process, as well as the IV created.
5501 It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the
5502 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic
5503 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which
5504 were generated.
5505 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested
5506 by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.
5507 The MACing key (client write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is
5508 flagged as valid for signing, verification and derivation operations.
5509 The other key (client write key) is typed according to information found in the template sent along with this
5510 mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption,
5511 decryption, and derivation operations.
5512 An IV (client write IV) will be generated and returned if the *ullVSizeInBits* field of the
5513 **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree
5514 with the value in the *ullVSizeInBits* field.
5515 Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
5516 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
5517 provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by
5518 the base key.
5519 Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS**
5520 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key
5521 handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-
5522 created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will
5523 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a
5524 buffer with sufficient space to hold any IV that will be returned.
5525 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
5526 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
5527 successful completion. However, since the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism
5528 returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the
5529 **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
5530 passed to **C_DeriveKey** is unnecessary, and should be a **NONE_PTR**.
5531 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

2.42 SP 800-108 Key Derivation

NIST SP800-108 defines three types of key derivation functions (KDF); a Counter Mode KDF, a Feedback Mode KDF and a Double Pipeline Mode KDF.

This section defines a unique mechanism for each type of KDF. These mechanisms can be used to derive one or more symmetric keys from a single base symmetric key.

The KDFs defined in SP800-108 are all built upon pseudo random functions (PRF). In general terms, the PRFs accept two pieces of input; a base key and some input data. The base key is taken from the *hBaseKey* parameter to **C_Derive**. The input data is constructed from an iteration variable (internally defined by the KDF/PRF) and the data provided in the CK_SP800_108_PRF_DATA_PARAM array that is part of the mechanism parameter.

Table 161, SP800-108 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SP800_108_COUNTER_KDF							✓
CKM_SP800_108_FEEDBACK_KDF							✓
CKM_SP800_108_DOUBLE_PIPELINE_KDF							✓

For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the minimum and maximum supported base key size in bits. Note, these mechanisms support multiple PRF types and key types; as such the values reported by *ulMinKeySize* and *ulMaxKeySize* specify the minimum and maximum supported base key size when all PRF and keys types are considered. For example, a Cryptoki implementation may support CKK_GENERIC_SECRET keys that can be as small as 8-bits in length and therefore *ulMinKeySize* could report 8-bits. However for an AES-CMAC PRF the base key must be of type CKK_AES and must be either 16-bytes, 24-bytes or 32-bytes in lengths and therefore the value reported by *ulMinKeySize* could be misleading. Depending on the PRF type selected, additional key size restrictions may apply.

2.42.1 Definitions

Mechanisms:

CKM_SP800_108_COUNTER_KDF
CKM_SP800_108_FEEDBACK_KDF
CKM_SP800_108_DOUBLE_PIPELINE_KDF

Data Field Types:

CK_SP800_108_ITERATION_VARIABLE
CK_SP800_108_COUNTER
CK_SP800_108_DKM_LENGTH
CK_SP800_108_BYTE_ARRAY

DKM Length Methods:

CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS
CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS

5568 **2.42.2 Mechanism Parameters**

5569 ◆ **CK_SP800_108_PRF_TYPE**

5570 The **CK_SP800_108_PRF_TYPE** field of the mechanism parameter is used to specify the type of PRF
5571 that is to be used. It is defined as follows:

5572

```
typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;
```

5573 The **CK_SP800_108_PRF_TYPE** field reuses the existing mechanisms definitions. The following table
5574 lists the supported PRF types:

5575 *Table 162, SP800-108 Pseudo Random Functions*

Pseudo Random Function Identifiers
CKM_SHA_1_HMAC
CKM_SHA224_HMAC
CKM_SHA256_HMAC
CKM_SHA384_HMAC
CKM_SHA512_HMAC
CKM_SHA3_224_HMAC
CKM_SHA3_256_HMAC
CKM_SHA3_384_HMAC
CKM_SHA3_512_HMAC
CKM_3DES_CMAC
CKM_AES_CMAC

5576

5577 ◆ **CK_PRF_DATA_TYPE**

5578 Each mechanism parameter contains an array of **CK_PRF_DATA_PARAM** structures. The
5579 **CK_PRF_DATA_PARAM** structure contains **CK_PRF_DATA_TYPE** field. The **CK_PRF_DATA_TYPE**
5580 field is used to identify the type of data identified by each **CK_PRF_DATA_PARAM** element in the array.
5581 Depending on the type of KDF used, some data field types are mandatory, some data field types are
5582 optional and some data field types are not allowed. These requirements are defined on a per-mechanism
5583 basis in the sections below. The **CK_PRF_DATA_TYPE** is defined as follows:

5584

```
typedef CK ULONG CK_PRF_DATA_TYPE;
```

5585 The following table lists all of the supported data field types:

5586 Table 163, SP800-108 PRF Data Field Types

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	Identifies the iteration variable defined internally by the KDF.
CK_SP800_108_COUNTER	Identifies an optional counter value represented as a binary string. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. The value of the counter is defined by the KDF's internal loop counter.
CK_SP800_108_DKM_LENGTH	Identifies the length in bits of the derived keying material (DKM) represented as a binary string. Exact formatting of the length value is defined by the CK_SP800_108_DKM_FORMAT structure.
CK_SP800_108_BYTE_ARRAY	Identifies a generic byte array of data. This data type can be used to provide "context", "label", "separator bytes" as well as any other type of encoding information required by the higher level protocol.

5587

5588 ◆ CK_PRF_DATA_PARAM

5589 **CK_PRF_DATA_PARAM** is used to define a segment of input for the PRF. Each mechanism parameter
 5590 supports an array of **CK_PRF_DATA_PARAM** structures. The **CK_PRF_DATA_PARAM** is defined as
 5591 follows:

```
5592     typedef struct CK_PRF_DATA_PARAM
5593     {
5594         CK_PRF_DATA_TYPE      type;
5595         CK_VOID_PTR           pValue;
5596         CK ULONG               ulValueLen;
5597     } CK_PRF_DATA_PARAM;
5598
5599     typedef CK_PRF_DATA_PARAM CK_PTR CK_PRF_DATA_PARAM_PTR
5600
```

5601 The fields of the **CK_PRF_DATA_PARAM** structure have the following meaning:

5602 **type** *defines the type of data pointed to by pValue*

5603 **pValue** *pointer to the data defined by type*

5604 **ulValueLen** *size of the data pointed to by pValue*

5605 If the **type** field of the **CK_PRF_DATA_PARAM** structure is set to
 5606 CK_SP800_108_ITERATION_VARIABLE, then **pValue** must be set the appropriate value for the KDF's
 5607 iteration variable type. For the Counter Mode KDF, **pValue** must be assigned a valid
 5608 CK_SP800_108_COUNTER_FORMAT_PTR and **ulValueLen** must be set to
 5609 sizeof(CK_SP800_108_COUNTER_FORMAT). For all other KDF types, **pValue** must be set to
 5610 NULL_PTR and **ulValueLen** must be set to 0.

5611

5612 If the **type** field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_COUNTER, then
 5613 **pValue** must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and **ulValueLen** must be
 5614 set to sizeof(CK_SP800_108_COUNTER_FORMAT).

5615

5616 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_DKM_LENGTH then
5617 *pValue* must be assigned a valid CK_SP800_108_DKM_FORMAT_PTR and *ulValueLen* must be set to
5618 sizeof(CK_SP800_108_DKM_FORMAT).

5619

5620 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_BYTE_ARRAY, then
5621 *pValue* must be assigned a valid CK_BYTE_PTR value and *ulValueLen* must be set to a non-zero length.

5622 ◆ **CK_SP800_108_COUNTER_FORMAT**

5623 **CK_SP800_108_COUNTER_FORMAT** is used to define the encoding format for a counter value. The
5624 **CK_SP800_108_COUNTER_FORMAT** is defined as follows:

```
5625     typedef struct CK_SP800_108_COUNTER_FORMAT
5626     {
5627         CK_BBOOL      bLittleEndian;
5628         CK ULONG      ulWidthInBits;
5629     } CK_SP800_108_COUNTER_FORMAT;
5630
5631     typedef CK_SP800_108_COUNTER_FORMAT CK_PTR
5632     CK_SP800_108_COUNTER_FORMAT_PTR
```

5633

5634 The fields of the CK_SP800_108_COUNTER_FORMAT structure have the following meaning:
5635 **bLittleEndian** *defines if the counter should be represented in Big Endian or Little*
5636 *Endian format*

5637 **ulWidthInBits** *defines the number of bits used to represent the counter value*

5638 ◆ **CK_SP800_108_DKM_LENGTH_METHOD**

5639 **CK_SP800_108_DKM_LENGTH_METHOD** is used to define how the DKM length value is calculated.
5640 The **CK_SP800_108_DKM_LENGTH_METHOD** type is defined as follows:

```
5641     typedef CK ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

5642 The following table lists all of the supported DKM Length Methods:

5643 *Table 164, SP800-108 DKM Length Methods*

DKM Length Method Identifier	Description
CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS	Specifies that the DKM length should be set to the sum of the length of all keys derived by this invocation of the KDF.
CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS	Specifies that the DKM length should be set to the sum of the length of all segments of output produced by the PRF by this invocation of the KDF.

5644

5645 ◆ **CK_SP800_108_DKM_LENGTH_FORMAT**

5646 **CK_SP800_108_DKM_LENGTH_FORMAT** is used to define the encoding format for the DKM length
5647 value. The **CK_SP800_108_DKM_LENGTH_FORMAT** is defined as follows:

```
5648     typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
```

```

5649      {
5650          CK_SP800_108_DKM_LENGTH_METHOD    dkmLengthMethod;
5651          CK_BBOOL                         bLittleEndian;
5652          CK ULONG                         ulWidthInBits;
5653      } CK_SP800_108_DKM_LENGTH_FORMAT;
5654
5655      typedef CK_SP800_108_DKM_LENGTH_FORMAT CK_PTR
5656      CK_SP800_108_DKM_LENGTH_FORMAT_PTR
5657
5658 The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:
5659     dkmLengthMethod      defines the method used to calculate the DKM length value
5660
5661             bLittleEndian      defines if the DKM length value should be represented in Big
5662                          Endian or LittleEndian format
5663             ulWidthInBits      defines the number of bits used to represent the DKM length value

```

5663 ◆ CK_DERIVED_KEY

5664 **CK_DERIVED_KEY** is used to define an additional key to be derived as well as provide a
 5665 **CK_OBJECT_HANDLE_PTR** to receive the handle for the derived keys. The **CK_DERIVED_KEY** is
 5666 defined as follows:

```

5667      typedef struct CK_DERIVED_KEY
5668      {
5669          CK_ATTRIBUTE_PTR      pTemplate;
5670          CK ULONG             ulAttributeCount;
5671          CK_OBJECT_HANDLE_PTR phKey;
5672      } CK_DERIVED_KEY;
5673
5674      typedef CK_DERIVED_KEY CK_PTR CK_DERIVED_KEY_PTR
5675

```

5676 The fields of the **CK_DERIVED_KEY** structure have the following meaning:
 5677 pTemplate pointer to a template that defines a key to derive
 5678 ulAttributeCount number of attributes in the template pointed to by pTemplate
 5679 phKey pointer to receive the handle for a derived key

5680 ◆ CK_SP800_108_KDF_PARAMS, CK_SP800_108_KDF_PARAMS_PTR

5681 **CK_SP800_108_KDF_PARAMS** is a structure that provides the parameters for the
 5682 **CKM_SP800_108_COUNTER_KDF** and **CKM_SP800_108_DOUBLE_PIPELINE_KDF** mechanisms.

```

5683
5684      typedef struct CK_SP800_108_KDF_PARAMS
5685      {
5686          CK_PRF_TYPE            prfType;
5687          CK ULONG               ulNumberOfDataParams;
5688          CK_PRF_DATA_PARAM_PTR pDataParams;
5689          CK ULONG               ulAdditionalDerivedKeys;

```

```

5690     CK_DERIVED_KEY           pAdditionalDerivedKeys;
5691 } CK_SP800_108_KDF_PARAMS;
5692
5693     typedef CK_SP800_108_KDF_PARAMS CK_PTR
5694     CK_SP800_108_KDF_PARAMS_PTR;
5695
5696 The fields of the CK_SP800_108_KDF_PARAMS structure have the following meaning:
5697     prfType      type of PRF
5698     ulNumberOfDataParams    number of elements in the array pointed to by pDataParams
5699     pDataParams        an array of CK_PRF_DATA_PARAM structures. The array defines
5700                           input parameters that are used to construct the "data" input to the
5701                           PRF.
5702     ulAdditionalDerivedKeys   number of additional keys that will be derived and the number of
5703                           elements in the array pointed to by pAdditionalDerivedKeys. If
5704                           pAdditionalDerivedKeys is set to NULL_PTR, this parameter must
5705                           be set to 0.
5706     pAdditionalDerivedKeys    an array of CK_DERIVED_KEY structures. If
5707                           ulAdditionalDerivedKeys is set to 0, this parameter must be set to
5708                           NULL_PTR

```

◆ **CK_SP800_108_FEEDBACK_KDF_PARAMS,**
CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR

5711 The **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure provides the parameters for the
5712 CKM_SP800_108_FEEDBACK_KDF mechanism. It is defined as follows:

```

5713     typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
5714     {
5715         CK_PRF_TYPE          prfType;
5716         CK ULONG             ulNumberOfDataParams;
5717         CK_PRF_DATA_PARAM_PTR pDataParams;
5718         CK ULONG             ulIVLen;
5719         CK_BYTE_PTR          pIV;
5720         CK ULONG             ulAdditionalDerivedKeys;
5721         CK_DERIVED_KEY       pAdditionalDerivedKeys;
5722     } CK_SP800_108_FEEDBACK_KDF_PARAMS;
5723
5724     typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR
5725     CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;
5726

```

5727 The fields of the **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure have the following meaning:
5728 prfType type of PRF

```

5729     ulNumberOfDataParams    number of elements in the array pointed to by pDataParams
5730     pDataParams        an array of CK_PRF_DATA_PARAM structures. The array defines
5731                           input parameters that are used to construct the "data" input to the
5732                           PRF.

```

5733	<i>ullIVLen</i>	<i>the length in bytes of the IV. If pIV is set to NULL_PTR, this parameter must be set to 0.</i>
5734		
5735	<i>pIV</i>	<i>an array of bytes to be used as the IV for the feedback mode KDF. This parameter is optional and can be set to NULL_PTR. If ullIVLen is set to 0, this parameter must be set to NULL_PTR.</i>
5736		
5737		
5738	<i>ulAdditionalDerivedKeys</i>	<i>number of additional keys that will be derived and the number of elements in the array pointed to by pAdditionalDerivedKeys. If pAdditionalDerivedKeys is set to NULL_PTR, this parameter must be set to 0.</i>
5739		
5740		
5741		
5742	<i>pAdditionalDerivedKeys</i>	<i>an array of CK_DERIVED_KEYS structures. If ulAdditionalDerivedKeys is set to 0, this parameter must be set to NULL_PTR.</i>
5743		
5744		

5745 2.42.3 Counter Mode KDF

5746 The SP800-108 Counter Mode KDF mechanism, denoted **CKM_SP800_108_COUNTER_KDF**,
 5747 represents the KDF defined SP800-108 section 5.1. **CKM_SP800_108_COUNTER_KDF** is a
 5748 mechanism for deriving one or more symmetric keys from a symmetric base key.

5749 It has a parameter, a **CK_SP800_108_KDF_PARAMS** structure.

5750 The following table lists the data field types that are supported for this KDF type and their meaning:

5751 *Table 165, Counter Mode data field requirements*

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable for this KDF type is a counter. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.
CK_SP800_108_COUNTER	This data field type is invalid for this KDF type.
CK_SP800_108_DKM_LENGTH	This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.
CK_SP800_108_BYTE_ARRAY	This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.

5752
 5753 SP800-108 limits the amount of derived keying material that can be produced by a Counter Mode KDF by
 5754 limiting the internal loop counter to $(2^r - 1)$, where "r" is the number of bits used to represent the counter.
 5755 Therefore the maximum number of bits that can be produced is $(2^r - 1)h$, where "h" is the length in bits of
 5756 the output of the selected PRF.

2.42.4 Feedback Mode KDF

The SP800-108 Feedback Mode KDF mechanism, denoted **CKM_SP800_108_FEEDBACK_KDF**, represents the KDF defined SP800-108 section 5.2. **CKM_SP800_108_FEEDBACK_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

Table 166, Feedback Mode data field requirements

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable is defined as K(i-1) in section 5.2 of SP800-108. The size, format and value of this data input is defined by the internal KDF structure and PRF output. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.
CK_SP800_108_COUNTER	This data field type is optional. This data field type identifies the location of the counter in the constructed PRF input data. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure. If specified, only one instance of this type may be specified.
CK_SP800_108_DKM_LENGTH	This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.
CK_SP800_108_BYTE_ARRAY	This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data. This standard does not restrict the number of instances of this data type.

5764

SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be produced is $(2^{32}-1)h$, where "h" is the length in bits of the output of the selected PRF.

2.42.5 Double Pipeline Mode KDF

The SP800-108 Double Pipeline Mode KDF mechanism, denoted **CKM_SP800_108_DOUBLE_PIPELINE_KDF**, represents the KDF defined SP800-108 section 5.3. **CKM_SP800_108_DOUBLE_PIPELINE_KDF** is a mechanism for deriving one or more symmetric keys from a symmetric base key.

It has a parameter, a **CK_SP800_108_KDF_PARAMS** structure.

The following table lists the data field types that are supported for this KDF type and their meaning:

5775 Table 167, Double Pipeline Mode data field requirements

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	<p>This data field type is mandatory.</p> <p>This data field type identifies the location of the iteration variable in the constructed PRF input data.</p> <p>The iteration variable is defined as A(i) in section 5.3 of SP800-108.</p> <p>The size, format and value of this data input is defined by the internal KDF structure and PRF output.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p>
CK_SP800_108_COUNTER	<p>This data field type is optional.</p> <p>This data field type identifies the location of the counter in the constructed PRF input data.</p> <p>Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_DKM_LENGTH	<p>This data field type is optional.</p> <p>This data field type identifies the location of the DKM length in the constructed PRF input data.</p> <p>Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure.</p> <p>If specified, only one instance of this type may be specified.</p>
CK_SP800_108_BYTE_ARRAY	<p>This data field type is optional.</p> <p>This data field type identifies the location and value of a byte array of data in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p>

5776

5777 SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode
 5778 KDF by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be
 5779 produced is $(2^{32}-1)h$, where "h" is the length in bits of the output of the selected PRF.

5780 The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method
 5781 used to construct the PRF input data; the data/values identified by the array of CK_PRF_DATA_PARAM
 5782 structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3,
 5783 the CK_SP800_108_ITERATION_VARIABLE and CK_SP800_108_COUNTER data field types are not
 5784 included in IV construction process. All other data field types are included in the construction process.

5785 2.42.6 Deriving Additional Keys

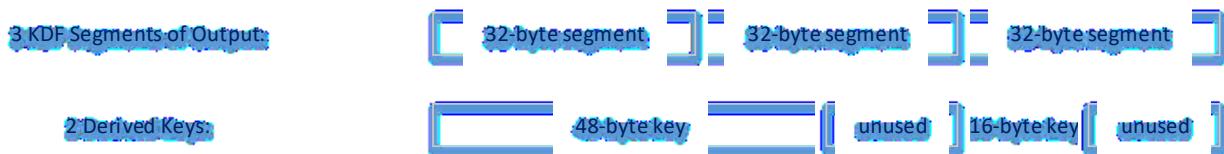
5786 The KDFs defined in this section can be used to derive more than one symmetric key from the base key.
 5787 The **C_Derive** function accepts one CK_ATTRIBUTE_PTR to define a single derived key and one
 5788 CK_OBJECT_HANDLE_PTR to receive the handle for the derived key.

5789 To derive additional keys, the mechanism parameter structure can be filled in with one or more
 5790 CK_DERIVED_KEY structures. Each structure contains a CK_ATTRIBUTE_PTR to define a derived key
 5791 and a CK_OBJECT_HANDLE_PTR to receive the handle for the additional derived keys. The key
 5792 defined by the **C_Derive** function parameters is always derived before the keys defined by the
 5793 CK_DERIVED_KEY array that is part of the mechanism parameter. The additional keys that are defined
 5794 by the CK_DERIVED_KEY array are derived in the order they are defined in the array. That is to say that
 5795 the derived keying material produced by the KDF is processed from left to right, and bytes are assigned

5796 first to the key defined by the **C_Derive** function parameters, and then bytes are assigned to the keys that
5797 are defined by the CK_DERIVED_KEY array in the order they are defined in the array.

5798 Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration
5799 will produce enough keying material for the key being derived. Other times, additional internal iterations
5800 are performed to produce multiple segments which are concatenated together to produce enough keying
5801 material for the derived key(s).

5802 When deriving multiple keys, no key can be created using part of a segment that was used for another
5803 key. All keys must be created from disjoint segments. For example, if the parameters are defined such
5804 that a 48-byte key (defined by the **C_Derive** function parameters) and a 16-byte key (defined by the
5805 content of CK_DERIVED_KEY) are to be derived using **CKM_SHA256_HMAC** as a PRF, three internal
5806 iterations of the KDF will be performed and three segments of PRF output will be produced. The first
5807 segment and half of the second segment will be used to create the 48-byte key and the third segment will
5808 be used to create the 16-byte key.



5809
5810 In the above example, if the CK_SP800_108_DKM_LENGTH data field type is specified with method
5811 CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, then the DKM length value will be 512 bits. If the
5812 CK_SP800_108_DKM_LENGTH data field type is specified with method
5813 CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS, then the DKM length value will be 768 bits.

5814 When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall
5815 be derived. If the failure was caused by the content of a specific key's template (ie the template defined
5816 by the content of *pTemplate*), the corresponding *phKey* value will be set to CK_HANDLE_INVALID to
5817 identify the offending template.

5819 2.42.7 Key Derivation Attribute Rules

5820 The **CKM_SP800_108_COUNTER_KDF**, **CKM_SP800_108_FEEDBACK_KDF** and
5821 **CKM_SP800_108_DOUBLE_PIPELINE_KDF** mechanisms have the following rules about key sensitivity
5822 and extractability:

- 5823 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key(s) can
5824 both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on
5825 some default value.
- 5826 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
5827 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
5828 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
5829 **CKA_SENSITIVE** attribute.
- 5830 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
5831 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
5832 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
5833 value from its **CKA_EXTRACTABLE** attribute.

5834 2.42.8 Constructing PRF Input Data

5835 SP800-108 defines the PRF input data for each KDF at a high level using terms like "label", "context",
5836 "separator", "counter"...etc. The value, formatting and order of the input data is not strictly defined by
5837 SP800-108, instead it is described as being defined by the "encoding scheme".

5838 To support any encoding scheme, these mechanisms construct the PRF input data from the array of
5839 CK_PRF_DATA_PARAM structures in the mechanism parameter. All of the values defined by the
5840 CK_PRF_DATA_PARAM array are concatenated in the order they are defined and passed in to the PRF
5841 as the data parameter.

5842 **2.42.8.1 Sample Counter Mode KDF**

5843 SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

5844 $\text{PRF}(\text{KI}, [i]_2 \parallel \text{Label} \parallel 0x00 \parallel \text{Context} \parallel [L]_2)$

5845 Section 5.1 does not define the number of bits used to represent the counter (the “r” value) or the DKM
 5846 length (the “L” value), so 16-bits is assumed for both cases. The following sample code shows how to
 5847 define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```

5848 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
5849
5850     CK_OBJECT_HANDLE hBaseKey;
5851     CK_OBJECT_HANDLE hDerivedKey;
5852     CK_ATTRIBUTE derivedKeyTemplate = { ... };
5853
5854     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
5855     CK ULONG ulLabelLen = sizeof(baLabel);
5856     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef}};
5857     CK ULONG ulContextLen = sizeof(baContext);
5858
5859     CK_SP800_108_COUNTER_FORMAT counterFormat = { 0, 16};
5860     CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};
5861
5862     CK_PRF_DATA_PARAM dataParams[] =
5863     {
5864         { CK_SP800_108_ITERATION_VARIABLE,
5865             &counterFormat, sizeof(counterFormat) },
5866         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
5867         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
5868         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
5869         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
5870     };
5871
5872     CK_SP800_108_KDF_PARAMS kdfParams =
5873     {
5874         CK_PRF_AES_CMAC,
5875         DIM(dataParams),
5876         &dataParams,
5877         0, /* no addition derived keys */
5878         NULL /* no addition derived keys */
5879     };
5880
5881     CK_MECHANISM = mechanism
5882     {
5883         CKM_FLEXIBLE_KDF,
5884         &kdfParams,
5885         sizeof(kdfParams)
5886     };
5887
5888     hBaseKey = GetBaseKeyHandle(.....);
5889
5890     rv = C_DeriveKey(
5891         hSession,
5892         &mechanism,
5893         hBaseKey,
5894         &derivedKeyTemplate,
5895         DIM(derivedKeyTemplate),

```

```
5896     &hDerivedKey);  
5897
```

5898 2.42.8.2 Sample SCP03 Counter Mode KDF

5899 The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:

5900 $\text{PRF} (K_1, \text{Label} \parallel 0x00 \parallel [L]_2 \parallel [i]_2 \parallel \text{Context})$

5901 SCP03 defines the number of bits used to represent the counter (the “r” value) and number of bits used to
5902 represent the DKM length (the “L” value) as 16-bits. The following sample code shows how to define this
5903 PRF input data using an array of CK_PRF_DATA_PARAM structures.

```
5904 #define DIM(a) (sizeof((a))/sizeof((a)[0]))  
5905  
5906     CK_OBJECT_HANDLE hBaseKey;  
5907     CK_OBJECT_HANDLE hDerivedKey;  
5908     CK_ATTRIBUTE derivedKeyTemplate = { ... };  
5909  
5910     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};  
5911     CK ULONG ulLabelLen = sizeof(baLabel);  
5912     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef}};  
5913     CK ULONG ulContextLen = sizeof(baContext);  
5914  
5915     CK_SP800_108_COUNTER_FORMAT counterFormat = { 0, 16};  
5916     CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};  
5917  
5918     CK_PRF_DATA_PARAM dataParams[] =  
5919     {  
5920         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },  
5921         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },  
5922         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },  
5923         { CK_SP800_108_ITERATION_VARIABLE,  
5924             &counterFormat, sizeof(counterFormat) },  
5925         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }  
5926     };  
5927  
5928     CK_SP800_108_KDF_PARAMS kdfParams =  
5929     {  
5930         CK_PRF_AES_CMAC,  
5931         DIM(dataParams),  
5932         &dataParams,  
5933         0, /* no addition derived keys */  
5934         NULL /* no addition derived keys */  
5935     };  
5936  
5937     CK_MECHANISM = mechanism  
5938     {  
5939         CKM_FLEXIBLE_KDF,  
5940         &kdfParams,  
5941         sizeof(kdfParams)  
5942     };  
5943  
5944     hBaseKey = GetBaseKeyHandle(.....);  
5945  
5946     rv = C_DeriveKey(  
5947         hSession,  
5948         &mechanism,
```

```

5949     hBaseKey,
5950     &derivedKeyTemplate,
5951     DIM(derivedKeyTemplate),
5952     &hDerivedKey);
5953

```

5954 2.42.8.3 Sample Feedback Mode KDF

5955 SP800-108 section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF input:

```
5956     PRF (Ki, K(i-1) {|| [i]₂ }|| Label || 0x00 || Context || [L]₂)
```

5957 Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM
 5958 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and
 5959 is included in this example. The following sample code shows how to define this PRF input data using an
 5960 array of CK_PRF_DATA_PARAM structures.

```

5961 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
5962
5963     CK_OBJECT_HANDLE hBaseKey;
5964     CK_OBJECT_HANDLE hDerivedKey;
5965     CK_ATTRIBUTE derivedKeyTemplate = { ... };
5966
5967     CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};
5968     CK ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);
5969     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
5970     CK ULONG ulLabelLen = sizeof(baLabel);
5971     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
5972     CK ULONG ulContextLen = sizeof(baContext);
5973
5974     CK_SP800_108_COUNTER_FORMAT counterFormat = { 0, 16 };
5975     CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};
5976
5977     CK_PRF_DATA_PARAM dataParams[] =
5978     {
5979         { CK_SP800_108_ITERATION_VARIABLE,
5980             &counterFormat, sizeof(counterFormat) },
5981         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
5982         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
5983         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
5984         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
5985     };
5986
5987     CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams =
5988     {
5989         CK_PRF_AES_CMAC,
5990         DIM(dataParams),
5991         &dataParams,
5992         ulFeedbackIVLen,
5993         pFeedbackIV,
5994         0, /* no addition derived keys */
5995         NULL /* no addition derived keys */
5996     };
5997
5998     CK_MECHANISM = mechanism
5999     {
6000         CKM_FLEXIBLE_KDF,
6001         &kdfParams,

```

```

6002     sizeof(kdfParams)
6003     } ;
6004
6005     hBaseKey = GetBaseKeyHandle(.....);
6006
6007     rv = C_DeriveKey(
6008         hSession,
6009         &mechanism,
6010         hBaseKey,
6011         &derivedKeyTemplate,
6012         DIM(derivedKeyTemplate),
6013         &hDerivedKey);
6014

```

6015 2.42.8.4 Sample Double-Pipeline Mode KDF

6016 SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following
 6017 PRF inputs:

```

6018     PRF (KI, A(i-1))
6019     PRF (KI, K(i-1) {|| [i]2 }|| Label || 0x00 || Context || [L]2)

```

6020 Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM
 6021 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it
 6022 is left out in this example. The following sample code shows how to define this PRF input data using an
 6023 array of CK_PRF_DATA_PARAM structures.

```

6024 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
6025
6026 CK_OBJECT_HANDLE hBaseKey;
6027 CK_OBJECT_HANDLE hDerivedKey;
6028 CK_ATTRIBUTE derivedKeyTemplate = { ... };
6029
6030 CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
6031 CK ULONG ulLabelLen = sizeof(baLabel);
6032 CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef}};
6033 CK ULONG ulContextLen = sizeof(baContext);
6034
6035 CK_SP800_108_DKM_FORMAT dkmFormat = {CK_SP800_108_SUM_OF_KEYS, 0, 16};
6036
6037 CK_PRF_DATA_PARAM dataParams[] =
6038 {
6039     { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
6040     { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
6041     { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
6042     { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
6043 };
6044
6045 CK_SP800_108_KDF_PARAMS kdfParams =
6046 {
6047     CK_PRF_AES_CMAC,
6048     DIM(dataParams),
6049     &dataParams,
6050     0, /* no addition derived keys */
6051     NULL /* no addition derived keys */
6052 };
6053
6054 CK_MECHANISM = mechanism
6055 {

```

```

6056     CKM_FLEXIBLE_KDF,
6057     &kdfParams,
6058     sizeof(kdfParams)
6059 } ;
6060
6061 hBaseKey = GetBaseKeyHandle(.....);
6062
6063 rv = C_DeriveKey(
6064     hSession,
6065     &mechanism,
6066     hBaseKey,
6067     &derivedKeyTemplate,
6068     DIM(derivedKeyTemplate),
6069     &hDerivedKey);
6070

```

2.43 Miscellaneous simple key derivation mechanisms

Table 168, *Miscellaneous simple key derivation Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CONCATENATE_BASE_AND_KEY							✓
CKM_CONCATENATE_BASE_AND_DATA							✓
CKM_CONCATENATE_DATA_AND_BASE							✓
CKM_XOR_BASE_AND_DATA							✓
CKM_EXTRACT_KEY_FROM_KEY							✓

2.43.1 Definitions

Mechanisms:

```

6075     CKM_CONCATENATE_BASE_AND_DATA
6076     CKM_CONCATENATE_DATA_AND_BASE
6077     CKM_XOR_BASE_AND_DATA
6078     CKM_EXTRACT_KEY_FROM_KEY
6079     CKM_CONCATENATE_BASE_AND_KEY

```

2.43.2 Parameters for miscellaneous simple key derivation mechanisms

- ◆ **CK_KEY_DERIVATION_STRING_DATA;**
- CK_KEY_DERIVATION_STRING_DATA_PTR**

6083 CK_KEY_DERIVATION_STRING_DATA provides the parameters for the
6084 CKM_CONCATENATE_BASE_AND_DATA, CKM_CONCATENATE_DATA_AND_BASE, and
6085 CKM_XOR_BASE_AND_DATA mechanisms. It is defined as follows:

```

6086     typedef struct CK_KEY_DERIVATION_STRING_DATA {
6087         CK_BYTE_PTR pData;
6088         CK_ULONG ulLen;

```

```
6089     } CK_KEY_DERIVATION_STRING_DATA;
6090
6091 The fields of the structure have the following meanings:
6092     pData      pointer to the byte string
6093     ulLen      length of the byte string
6094 CK_KEY_DERIVATION_STRING_DATA_PTR is a pointer to a
6095 CK_KEY_DERIVATION_STRING_DATA.
```

6096 ♦ CK_EXTRACT_PARAMS; CK_EXTRACT_PARAMS_PTR

```
6097 CK_EXTRACT_PARAMS provides the parameter to the CKM_EXTRACT_KEY_FROM_KEY
6098 mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is
6099 defined as follows:
```

```
6100     typedef CK ULONG CK_EXTRACT_PARAMS;
```

```
6101
```

```
6102 CK_EXTRACT_PARAMS_PTR is a pointer to a CK_EXTRACT_PARAMS.
```

6103 2.43.3 Concatenation of a base key and another key

```
6104 This mechanism, denoted CKM_CONCATENATE_BASE_AND_KEY, derives a secret key from the
6105 concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys
6106 specified are concatenated together in a buffer.
```

```
6107 This mechanism takes a parameter, a CK_OBJECT_HANDLE. This handle produces the key value
6108 information which is appended to the end of the base key's value information (the base key is the key
6109 whose handle is supplied as an argument to C_DeriveKey).
```

```
6110 For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF,
6111 then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.
```

- 6112 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
6113 generic secret key. Its length will be equal to the sum of the lengths of the values of the two original
6114 keys.
- 6115 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
6116 will be a generic secret key of the specified length.
- 6117 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
6118 length. If it does, then the key produced by this mechanism will be of the type specified in the
6119 template. If it doesn't, an error will be returned.
- 6120 • If both a key type and a length are provided in the template, the length must be compatible with that
6121 key type. The key produced by this mechanism will be of the specified type and length.

```
6122 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
6123 properly.
```

```
6124 If the requested type of key requires more bytes than are available by concatenating the two original keys'
6125 values, an error is generated.
```

```
6126 This mechanism has the following rules about key sensitivity and extractability:
```

- 6127 • If either of the two original keys has its CKA_SENSITIVE attribute set to CK_TRUE, so does the
6128 derived key. If not, then the derived key's CKA_SENSITIVE attribute is set either from the supplied
6129 template or from a default value.
- 6130 • Similarly, if either of the two original keys has its CKA_EXTRACTABLE attribute set to CK_FALSE,
6131 so does the derived key. If not, then the derived key's CKA_EXTRACTABLE attribute is set either
6132 from the supplied template or from a default value.

- 6133 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if both of the
6134 original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to CK_TRUE.
6135 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
6136 both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to CK_TRUE.

6137 **2.43.4 Concatenation of a base key and data**

6138 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by
6139 concatenating data onto the end of a specified secret key.

6140 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
6141 specifies the length and value of the data which will be appended to the base key to derive another key.

6142 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
6143 the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 6144 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
6145 generic secret key. Its length will be equal to the sum of the lengths of the value of the original key
6146 and the data.
- 6147 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
6148 will be a generic secret key of the specified length.
- 6149 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
6150 length. If it does, then the key produced by this mechanism will be of the type specified in the
6151 template. If it doesn't, an error will be returned.
- 6152 • If both a key type and a length are provided in the template, the length must be compatible with that
6153 key type. The key produced by this mechanism will be of the specified type and length.

6154 If a DES, DES2, DESS, or CDMF key is derived with this mechanism, the parity bits of the key will be set
6155 properly.

6156 If the requested type of key requires more bytes than are available by concatenating the original key's
6157 value and the data, an error is generated.

6158 This mechanism has the following rules about key sensitivity and extractability:

- 6159 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
6160 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
6161 default value.
- 6162 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
6163 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
6164 supplied template or from a default value.
- 6165 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
6166 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 6167 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
6168 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6169 **2.43.5 Concatenation of data and a base key**

6170 This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by
6171 prepending data to the start of a specified secret key.

6172 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
6173 specifies the length and value of the data which will be prepended to the base key to derive another key.

6174 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
6175 the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.

- 6176 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
6177 generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the
6178 original key.

- 6179 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
6180 will be a generic secret key of the specified length.
6181 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
6182 length. If it does, then the key produced by this mechanism will be of the type specified in the
6183 template. If it doesn't, an error will be returned.
6184 • If both a key type and a length are provided in the template, the length must be compatible with that
6185 key type. The key produced by this mechanism will be of the specified type and length.

6186 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
6187 properly.

6188 If the requested type of key requires more bytes than are available by concatenating the data and the
6189 original key's value, an error is generated.

6190 This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
6192 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
6193 default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
6195 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
6196 supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
6198 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
6200 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6201 2.43.6 XORing of a key and data

6202 XORing key derivation, denoted **CKM_XOR_BASE_AND_DATA**, is a mechanism which provides the
6203 capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle
6204 and some data.

6205 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
6206 specifies the data with which to XOR the original key's value.

6207 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
6208 the value of the derived key will be taken from a buffer containing the string 0x88888888.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a
6210 generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of
6211 the original key.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism
6213 will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined
6215 length. If it does, then the key produced by this mechanism will be of the type specified in the
6216 template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that
6218 key type. The key produced by this mechanism will be of the specified type and length.

6219 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
6220 properly.

6221 If the requested type of key requires more bytes than are available by taking the shorter of the data and
6222 the original key's value, an error is generated.

6223 This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
6225 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
6226 default value.

- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

2.43.7 Extraction of one key from another key

Extraction of one key from another key, denoted **CKM_EXTRACT_KEY_FROM_KEY**, is a mechanism which provides the capability of creating one secret key from the bits of another secret key.

This mechanism has a parameter, a **CK_EXTRACT_PARAMS**, which specifies which bit of the original key should be used as the first bit of the newly-derived key.

We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the value of the parameter to the **CKM_EXTRACT_KEY_FROM_KEY** mechanism is 21).

1. We write the key's value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this binary string as holding the 32 bits of the key, labeled as b0, b1, ..., b31.
2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We obtain the binary string 1001 0101 0010 0110.
3. The value of the new key is thus 0x9526.

Note that when constructing the value of the derived key, it is permissible to wrap around the end of the binary string representing the original key's value.

If the original key used in this process is sensitive, then the derived key must also be sensitive for the derivation to succeed.

- If no length or key type is provided in the template, then an error will be returned.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism will be a generic secret key of the specified length.
- If no length is provided in the template, but a key type is, then that key type must have a well-defined length. If it does, then the key produced by this mechanism will be of the type specified in the template. If it doesn't, an error will be returned.
- If both a key type and a length are provided in the template, the length must be compatible with that key type. The key produced by this mechanism will be of the specified type and length.

If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set properly.

If the requested type of key requires more bytes than the original key has, an error is generated.

This mechanism has the following rules about key sensitivity and extractability:

- If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a default value.
- Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the supplied template or from a default value.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6273 2.44 CMS

6274 Table 169, CMS Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CMS_SIG		✓	✓				

6275 2.44.1 Definitions

6276 Mechanisms:

6277 CKM_CMS_SIG

6278 2.44.2 CMS Signature Mechanism Objects

6279 These objects provide information relating to the CKM_CMS_SIG mechanism. CKM_CMS_SIG
6280 mechanism object attributes represent information about supported CMS signature attributes in the token.
6281 They are only present on tokens supporting the **CKM_CMS_SIG** mechanism, but must be present on
6282 those tokens.

6283 Table 170, CMS Signature Mechanism Object Attributes

Attribute	Data type	Meaning
CKA_REQUIRED_CMS_ATTRIBUTES	Byte array	Attributes the token always will include in the set of CMS signed attributes
CKA_DEFAULT_CMS_ATTRIBUTES	Byte array	Attributes the token will include in the set of CMS signed attributes in the absence of any attributes specified by the application
CKA_SUPPORTED_CMS_ATTRIBUTES	Byte array	Attributes the token may include in the set of CMS signed attributes upon request by the application

6284 The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying
6285 values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-
6286 encoded. The list of attributes is defined in ASN.1 as:

```
6287     Attributes ::= SET SIZE (1..MAX) OF Attribute
6288     Attribute ::= SEQUENCE {
6289         attrType   OBJECT IDENTIFIER,
6290         attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER
6291             OPTIONAL
6292     }
```

6293 The client may not set any of the attributes.

6294 2.44.3 CMS mechanism parameters

- **CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR**

6295 **CK_CMS_SIG_PARAMS** is a structure that provides the parameters to the **CKM_CMS_SIG** mechanism.
6296 It is defined as follows:

```
6298     typedef struct CK_CMS_SIG_PARAMS {
```

```

6299    CK_OBJECT_HANDLE           certificateHandle;
6300    CK_MECHANISM_PTR          pSigningMechanism;
6301    CK_MECHANISM_PTR          pDigestMechanism;
6302    CK_UTF8CHAR_PTR           pContentType;
6303    CK_BYTE_PTR                pRequestedAttributes;
6304    CK ULONG                   ulRequestedAttributesLen;
6305    CK_BYTE_PTR                pRequiredAttributes;
6306    CK ULONG                   ulRequiredAttributesLen;
6307 } CK_CMS_SIG_PARAMS;
6308
6309 The fields of the structure have the following meanings:
6310     certificateHandle          Object handle for a certificate associated with the signing key. The token may use information from this certificate to identify the signer in the SignerInfo result value. CertificateHandle may be NULL_PTR if the certificate is not available as a PKCS #11 object or if the calling application leaves the choice of certificate completely to the token.
6311
6312
6313
6314
6315
6316     pSigningMechanism          Mechanism to use when signing a constructed CMS SignedAttributes value. E.g. CKM_SHA1_RSA_PKCS.
6317
6318     pDigestMechanism          Mechanism to use when digesting the data. Value shall be NULL_PTR when the digest mechanism to use follows from the pSigningMechanism parameter.
6319
6320
6321     pContentType               NULL-terminated string indicating complete MIME Content-type of message to be signed; or the value NULL_PTR if the message is a MIME object (which the token can parse to determine its MIME Content-type if required). Use the value "application/octet-stream" if the MIME type for the message is unknown or undefined. Note that the pContentType string shall conform to the syntax specified in RFC 2045, i.e. any parameters needed for correct presentation of the content by the token (such as, for example, a non-default "charset") must be present. The token must follow rules and procedures defined in RFC 2045 when presenting the content.
6322
6323
6324
6325
6326
6327
6328
6329
6330
6331     pRequestedAttributes       Pointer to DER-encoded list of CMS Attributes the caller requests to be included in the signed attributes. Token may freely ignore this list or modify any supplied values.
6332
6333
6334     ulRequestedAttributesLen   Length in bytes of the value pointed to by pRequestedAttributes
6335
6336
6337
6338
6339
6340
6341
6342     pRequiredAttributes        Pointer to DER-encoded list of CMS Attributes (with accompanying values) required to be included in the resulting signed attributes. Token must not modify any supplied values. If the token does not support one or more of the attributes, or does not accept provided values, the signature operation will fail. The token will use its own default attributes when signing if both the pRequestedAttributes and pRequiredAttributes field are set to NULL_PTR.
6343
6344     ulRequiredAttributesLen   Length in bytes, of the value pointed to by pRequiredAttributes.

```

6343 2.44.4 CMS signatures

6344 The CMS mechanism, denoted **CKM_CMS_SIG**, is a multi-purpose mechanism based on the structures
6345 defined in PKCS #7 and RFC 2630. It supports single- or multiple-part signatures with and without
6346 message recovery. The mechanism is intended for use with, e.g., PTDs (see MeT-PTD) or other capable
6347 tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this value.
6348 The content of the **SignedAttributes** value is decided by the token, however the caller can suggest some
6349 attributes in the parameter *pRequestedAttributes*. The caller can also require some attributes to be
6350 present through the parameters *pRequiredAttributes*. The signature is computed in accordance with the
6351 parameter *pSigningMechanism*.

6352 When this mechanism is used in successful calls to **C_Sign** or **C_SignFinal**, the *pSignature* return value
6353 will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a
6354 complete definition of all fields and types, see RFC 2630):

```
6355     SignerInfo ::= SEQUENCE {
6356         version CMSVersion,
6357         sid SignerIdentifier,
6358         digestAlgorithm DigestAlgorithmIdentifier,
6359         signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
6360         signatureAlgorithm SignatureAlgorithmIdentifier,
6361         signature SignatureValue,
6362         unsignedAttrs [1] IMPLICIT UnsignedAttributes
6363         OPTIONAL }
```

6364 The *certificateHandle* parameter, when set, helps the token populate the **sid** field of the **SignerInfo** value.
6365 If *certificateHandle* is NULL_PTR the choice of a suitable certificate reference in the **SignerInfo** result
6366 value is left to the token (the token could, e.g., interact with the user).

6367 This mechanism shall not be used in calls to **C_Verify** or **C_VerifyFinal** (use the *pSigningMechanism*
6368 mechanism instead).

6369 For the *pRequiredAttributes* field, the token may have to interact with the user to find out whether to
6370 accept a proposed value or not. The token should never accept any proposed attribute values without
6371 some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings
6372 and not direct interaction). If a user rejects proposed values, or the signature request as such, the value
6373 CKR_FUNCTION_REJECTED shall be returned.

6374 When possible, applications should use the **CKM_CMS_SIG** mechanism when generating CMS-
6375 compatible signatures rather than lower-level mechanisms such as **CKM_SHA1_RSA_PKCS**. This is
6376 especially true when the signatures are to be made on content that the token is able to present to a user.
6377 Exceptions may include those cases where the token does not support a particular signing attribute. Note
6378 however that the token may refuse usage of a particular signature key unless the content to be signed is
6379 known (i.e. the **CKM_CMS_SIG** mechanism is used).

6380 When a token does not have presentation capabilities, the PKCS #11-aware application may avoid
6381 sending the whole message to the token by electing to use a suitable signature mechanism (e.g.
6382 **CKM_RSA_PKCS**) as the *pSigningMechanism* value in the **CK_CMS_SIG_PARAMS** structure, and
6383 digesting the message itself before passing it to the token.

6384 PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to
6385 provide messages to be signed by the token in a format possible for the token to present to the user.
6386 Tokens that receive multipart MIME-messages for which only certain parts are possible to present may
6387 fail the signature operation with a return value of **CKR_DATA_INVALID**, but may also choose to add a
6388 signing attribute indicating which parts of the message were possible to present.

6389 2.45 Blowfish

6390 Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times.
6391 The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex

6392 initialization phase required before any encryption can take place, the actual encryption of data is very
6393 efficient on large microprocessors.

6394

6395 *Table 171, Blowfish Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_BLOWFISH_CBC	✓					✓	
CKM_BLOWFISH_CBC_PAD	✓					✓	

6396 **2.45.1 Definitions**

6397 This section defines the key type “CKK_BLOWFISH” for type CK_KEY_TYPE as used in the
6398 CKA_KEY_TYPE attribute of key objects.

6399 Mechanisms:

6400 CKM_BLOWFISH_KEY_GEN
6401 CKM_BLOWFISH_CBC
6402 CKM_BLOWFISH_CBC_PAD

6403 **2.45.2 BLOWFISH secret key objects**

6404 Blowfish secret key objects (object class CKO_SECRET_KEY, key type CKK_BLOWFISH) hold Blowfish
6405 keys. The following table defines the Blowfish secret key object attributes, in addition to the common
6406 attributes defined for this object class:

6407 *Table 172, BLOWFISH Secret Key Object*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value the key can be any length up to 448 bits. Bit length restricted to a byte array.
CKA_VALUE_LEN ^{2,3}	CK ULONG	Length in bytes of key value

6408 - Refer to [PKCS11-Base] table 11 for footnotes

6409 The following is a sample template for creating an Blowfish secret key object:

```
6410     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
6411     CK_KEY_TYPE keyType = CKK_BLOWFISH;
6412     CK_UTF8CHAR label[] = "A blowfish secret key object";
6413     CK_BYTE value[16] = {...};
6414     CK_BBOOL true = CK_TRUE;
6415     CK_ATTRIBUTE template[] = {
6416         {CKA_CLASS, &class, sizeof(class)},
6417         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6418         {CKA_TOKEN, &true, sizeof(true)},
6419         {CKA_LABEL, label, sizeof(label)-1},
6420         {CKA_ENCRYPT, &true, sizeof(true)},
6421         {CKA_VALUE, value, sizeof(value)}
```

6422 } ;

6423 2.45.3 Blowfish key generation

6424 The Blowfish key generation mechanism, denoted **CKM_BLOWFISH_KEY_GEN**, is a key generation
6425 mechanism Blowfish.

6426 It does not have a parameter.

6427 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN**
6428 attribute of the template for the key.

6429 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
6430 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
6431 supports) may be specified in the template for the key, or else are assigned default initial values.

6432 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6433 specify the supported range of key sizes in bytes.

6434 2.45.4 Blowfish-CBC

6435 Blowfish-CBC, denoted **CKM_BLOWFISH_CBC**, is a mechanism for single- and multiple-part encryption
6436 and decryption; key wrapping; and key unwrapping.

6437 It has a parameter, a 8-byte initialization vector.

6438 This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value
6439 of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size
6440 minus one null bytes so that the resulting length is a multiple of the block size. The output data is the
6441 same length as the padded input data. It does not wrap the key type, key length, or any other information
6442 about the key; the application must convey these separately.

6443 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
6444 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
6445 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
6446 attribute of the new key; other attributes required by the key type must be specified in the template.

6447 Constraints on key types and the length of data are summarized in the following table:

6448 Table 173, BLOWFISH-CBC: Key and Data Length

Function	Key type	Input Length	Output Length
C_Encrypt	BLOWFISH	Multiple of block size	Same as input length
C_Decrypt	BLOWFISH	Multiple of block size	Same as input length
C_WrapKey	BLOWFISH	Any	Input length rounded up to multiple of the block size
C_UnwrapKey	BLOWFISH	Multiple of block size	Determined by type of key being unwrapped or CKA_VALUE_LEN

6449 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6450 specify the supported range of BLOWFISH key sizes, in bytes.

6451 2.45.5 Blowfish-CBC with PKCS padding

6452 Blowfish-CBC-PAD, denoted **CKM_BLOWFISH_CBC_PAD**, is a mechanism for single- and multiple-part
6453 encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block
6454 cipher padding method detailed in PKCS #7.

6455 It has a parameter, a 8-byte initialization vector.

6456 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
6457 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
6458 the **CKA_VALUE_LEN** attribute.

6459 The entries in the table below for data length constraints when wrapping and unwrapping keys do not
6460 apply to wrapping and unwrapping private keys.

6461 Constraints on key types and the length of data are summarized in the following table:

6462

6463 *Table 174, BLOWFISH-CBC with PKCS Padding: Key and Data Length*

Function	Key type	Input Length	Output Length
C_Encrypt	BLOWFISH	Any	Input length rounded up to multiple of the block size
C_Decrypt	BLOWFISH	Multiple of block size	Between 1 and block length block size bytes shorter than input length
C_WrapKey	BLOWFISH	Any	Input length rounded up to multiple of the block size
C_UnwrapKey	BLOWFISH	Multiple of block size	Between 1 and block length block size bytes shorter than input length

6464 **2.46 Twofish**

6465 Ref. <https://www.schneier.com/twofish.html>

6466 **2.46.1 Definitions**

6467 This section defines the key type “CKK_TWOFISH” for type CK_KEY_TYPE as used in the
6468 CKA_KEY_TYPE attribute of key objects.

6469 Mechanisms:

6470 CKM_TWOFISH_KEY_GEN

6471 CKM_TWOFISH_CBC

6472 CKM_TWOFISH_CBC_PAD

6473

6474 **2.46.2 Twofish secret key objects**

6475 Twofish secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_TWOFISH**) hold Twofish
6476 keys. The following table defines the Twofish secret key object attributes, in addition to the common
6477 attributes defined for this object class:

6478 *Table 175, Twofish Secret Key Object*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value 128-, 192-, or 256-bit key
CKA_VALUE_LEN ^{2,3}	CK ULONG	Length in bytes of key value

6479 - Refer to [PKCS11-Base] table 11 for footnotes

6480 The following is a sample template for creating an TWOIFSH secret key object:

```
6481     CK_OBJECT_CLASS class = CKO_SECRET_KEY;  
6482     CK_KEY_TYPE keyType = CKK_TWOFISH;  
6483     CK_UTF8CHAR label[] = "A twofish secret key object";  
6484     CK_BYTE value[16] = {...};  
6485     CK_BBOOL true = CK_TRUE;
```

```

6486     CK_ATTRIBUTE template[] = {
6487         { CKA_CLASS, &class, sizeof(class) },
6488         { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
6489         { CKA_TOKEN, &true, sizeof(true) },
6490         { CKA_LABEL, label, sizeof(label)-1 },
6491         { CKA_ENCRYPT, &true, sizeof(true) },
6492         { CKA_VALUE, value, sizeof(value) }
6493     };

```

6494 **2.46.3 Twofish key generation**

6495 The Twofish key generation mechanism, denoted **CKM_TWOFISH_KEY_GEN**, is a key generation
 6496 mechanism Twofish.

6497 It does not have a parameter.

6498 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN**
 6499 attribute of the template for the key.

6500 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 6501 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
 6502 supports) may be specified in the template for the key, or else are assigned default initial values.

6503 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 6504 specify the supported range of key sizes, in bytes.

6505 **2.46.4 Twofish -CBC**

6506 Twofish-CBC, denoted **CKM_TWOFISH_CBC**, is a mechanism for single- and multiple-part encryption
 6507 and decryption; key wrapping; and key unwrapping.

6508 It has a parameter, a 16-byte initialization vector.

6509 **2.46.5 Twofish-CBC with PKCS padding**

6510 Twofish-CBC-PAD, denoted **CKM_TWOFISH_CBC_PAD**, is a mechanism for single- and multiple-part
 6511 encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block
 6512 cipher padding method detailed in PKCS #7.

6513 It has a parameter, a 16-byte initialization vector.

6514 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
 6515 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
 6516 the **CKA_VALUE_LEN** attribute.

6517 **2.47 CAMELLIA**

6518 Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.
 6519 Camellia is described e.g. in IETF RFC 3713.

6520 *Table 176, Camellia Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Deriv
CKM_CAMELLIA_KEY_GEN					✓		

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CAMELLIA_ECB	✓					✓	
CKM_CAMELLIA_CBC	✓					✓	
CKM_CAMELLIA_CBC_PAD	✓					✓	
CKM_CAMELLIA_MAC_GENERAL		✓					
CKM_CAMELLIA_MAC		✓					
CKM_CAMELLIA_ECB_ENCRYPT_DATA							✓
CKM_CAMELLIA_CBC_ENCRYPT_DATA							✓

2.47.1 Definitions

This section defines the key type “CKK_CAMELLIA” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_CAMELLIA_KEY_GEN
- CKM_CAMELLIA_ECB
- CKM_CAMELLIA_CBC
- CKM_CAMELLIA_MAC
- CKM_CAMELLIA_MAC_GENERAL
- CKM_CAMELLIA_CBC_PAD

2.47.2 Camellia secret key objects

Camellia secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_CAMELLIA**) hold Camellia keys. The following table defines the Camellia secret key object attributes, in addition to the common attributes defined for this object class:

Table 177, Camellia Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN ^{2,3,6}	CK ULONG	Length in bytes of key value

- Refer to [PKCS11-Base] table 11 for footnotes.

The following is a sample template for creating a Camellia secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CAMELLIA;
CK_UTF8CHAR label[] = "A Camellia secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
```

```
6544     { CKA_CLASS, &class, sizeof(class) },
6545     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
6546     { CKA_TOKEN, &true, sizeof(true) },
6547     { CKA_LABEL, label, sizeof(label)-1 },
6548     { CKA_ENCRYPT, &true, sizeof(true) },
6549     { CKA_VALUE, value, sizeof(value) }
6550 };
```

6551 **2.47.3 Camellia key generation**

6552 The Camellia key generation mechanism, denoted **CKM_CAMELLIA_KEY_GEN**, is a key generation
6553 mechanism for Camellia.

6554 It does not have a parameter.

6555 The mechanism generates Camellia keys with a particular length in bytes, as specified in the
6556 **CKA_VALUE_LEN** attribute of the template for the key.

6557 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
6558 key. Other attributes supported by the Camellia key type (specifically, the flags indicating which functions
6559 the key supports) may be specified in the template for the key, or else are assigned default initial values.

6560 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6561 specify the supported range of Camellia key sizes, in bytes.

6562 **2.47.4 Camellia-ECB**

6563 Camellia-ECB, denoted **CKM_CAMELLIA_ECB**, is a mechanism for single- and multiple-part encryption
6564 and decryption; key wrapping; and key unwrapping, based on Camellia and electronic codebook mode.

6565 It does not have a parameter.

6566 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
6567 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
6568 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
6569 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
6570 length as the padded input data. It does not wrap the key type, key length, or any other information about
6571 the key; the application must convey these separately.

6572 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
6573 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
6574 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
6575 attribute of the new key; other attributes required by the key type must be specified in the template.

6576 Constraints on key types and the length of data are summarized in the following table:

6577 Table 178, Camellia-ECB: Key and Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of block size	
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

6578 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 6579 specify the supported range of Camellia key sizes, in bytes.

6580 2.47.5 Camellia-CBC

6581 Camellia-CBC, denoted **CKM_CAMELLIA_CBC**, is a mechanism for single- and multiple-part encryption
 6582 and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.
 6583 It has a parameter, a 16-byte initialization vector.

6584 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
 6585 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
 6586 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
 6587 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
 6588 length as the padded input data. It does not wrap the key type, key length, or any other information about
 6589 the key; the application must convey these separately.

6590 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
 6591 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
 6592 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
 6593 attribute of the new key; other attributes required by the key type must be specified in the template.

6594 Constraints on key types and the length of data are summarized in the following table:

6595 Table 179, Camellia-CBC: Key and Data Length

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_CAMELLIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of the block size	
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

6596 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 6597 specify the supported range of Camellia key sizes, in bytes.

6598 2.47.6 Camellia-CBC with PKCS padding

6599 Camellia-CBC with PKCS padding, denoted **CKM_CAMELLIA_CBC_PAD**, is a mechanism for single-
6600 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia;
6601 cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

6602 It has a parameter, a 16-byte initialization vector.

6603 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
6604 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified
6605 for the **CKA_VALUE_LEN** attribute.

6606 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
6607 Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section
6608 TBA for details). The entries in the table below for data length constraints when wrapping and
6609 unwrapping keys do not apply to wrapping and unwrapping private keys.

6610 Constraints on key types and the length of data are summarized in the following table:

6611 *Table 180, Camellia-CBC with PKCS Padding: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_CAMELLIA	any	input length rounded up to multiple of the block size
C_Decrypt	CKK_CAMELLIA	multiple of block size	between 1 and block length bytes shorter than input length
C_WrapKey	CKK_CAMELLIA	any	input length rounded up to multiple of the block size
C_UnwrapKey	CKK_CAMELLIA	multiple of block size	between 1 and block length bytes shorter than input length

6612 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6613 specify the supported range of Camellia key sizes, in bytes.

6614

6615 2.47.7 CAMELLIA with Counter mechanism parameters

6616 ♦ **CK_CAMELLIA_CTR_PARAMS; CK_CAMELLIA_CTR_PARAMS_PTR**

6617 **CK_CAMELLIA_CTR_PARAMS** is a structure that provides the parameters to the
6618 **CKM_CAMELLIA_CTR** mechanism. It is defined as follows:

```
6619     typedef struct CK_CAMELLIA_CTR_PARAMS {
6620         CK ULONG ulCounterBits;
6621         CK_BYTE cb[16];
6622     } CK_CAMELLIA_CTR_PARAMS;
```

6624 *ulCounterBits* specifies the number of bits in the counter block (*cb*) that shall be incremented. This
6625 number shall be such that $0 < \text{ulCounterBits} \leq 128$. For any values outside this range the mechanism
6626 shall return **CKR_MECHANISM_PARAM_INVALID**.

6627 It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter
6628 bits are the least significant bits of the counter block (*cb*). They are a big-endian value usually starting
6629 with 1. The rest of '*cb*' is for the nonce, and maybe an optional IV.

6630 E.g. as defined in [RFC 3686]:

```

6631      0           1           2           3
6632      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
6633 +-----+-----+-----+-----+-----+-----+
6634 |                   Nonce                   |
6635 +-----+-----+-----+-----+-----+-----+-----+
6636 |                   Initialization Vector (IV)   |
6637 |                   |
6638 +-----+-----+-----+-----+-----+-----+-----+
6639 |                   Block Counter            |
6640 +-----+-----+-----+-----+-----+-----+-----+

```

6641

6642 This construction permits each packet to consist of up to $2^{32}-1$ blocks = 4,294,967,295 blocks =
6643 68,719,476,720 octets.

6644 **CK_CAMELLIA_CTR_PARAMS_PTR** is a pointer to a **CK_CAMELLIA_CTR_PARAMS**.

6645

6646 2.47.8 General-length Camellia-MAC

6647 General-length Camellia -MAC, denoted CKM_CAMELLIA_MAC_GENERAL, is a mechanism for single-
6648 and multiple-part signatures and verification, based on Camellia and data authentication as defined
6649 in.[CAMELLIA]

6650 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
6651 desired from the mechanism.

6652 The output bytes from this mechanism are taken from the start of the final Camellia cipher block produced
6653 in the MACing process.

6654 Constraints on key types and the length of data are summarized in the following table:

6655 *Table 181, General-length Camellia-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	1-block size, as specified in parameters
C_Verify	CKK_CAMELLIA	any	1-block size, as specified in parameters

6656 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6657 specify the supported range of Camellia key sizes, in bytes.

6658 2.47.9 Camellia-MAC

6659 Camellia-MAC, denoted by **CKM_CAMELLIA_MAC**, is a special case of the general-length Camellia-
6660 MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in
6661 length.

6662 It does not have a parameter.

6663 Constraints on key types and the length of data are summarized in the following table:

6664 *Table 182, Camellia-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	$\frac{1}{2}$ block size (8 bytes)
C_Verify	CKK_CAMELLIA	any	$\frac{1}{2}$ block size (8 bytes)

6665 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6666 specify the supported range of Camellia key sizes, in bytes.

6667 2.48 Key derivation by data encryption - Camellia

6668 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
6669 They are for use with the C_DeriveKey function.

6670 2.48.1 Definitions

6671 Mechanisms:

6672 CKM_CAMELLIA_ECB_ENCRYPT_DATA
6673 CKM_CAMELLIA_CBC_ENCRYPT_DATA

6674

```
6675     typedef struct CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS {  
6676         CK_BYTE          iv[16];  
6677         CK_BYTE_PTR      pData;  
6678         CK ULONG         length;  
6679     } CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS;  
6680  
6681     typedef CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
6682             CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6683 2.48.2 Mechanism Parameters

6684 Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

6685 *Table 183, Mechanism Parameters for Camellia-based key derivation*

CKM_CAMELLIA_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_CAMELLIA_CBC_ENCRYPT_DATA	Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6686

6687 2.49 ARIA

6688 ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is
6689 described in NSRI "Specification of ARIA".

6690 *Table 184, ARIA Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ARIA_KEY_GEN					✓		
CKM_ARIA_ECB	✓					✓	
CKM_ARIA_CBC	✓					✓	
CKM_ARIA_CBC_PAD	✓					✓	

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_ARIA_MAC_GENERAL		✓					
CKM_ARIA_MAC		✓					
CKM_ARIA_ECB_ENCRYPT_DATA							✓
CKM_ARIA_CBC_ENCRYPT_DATA							✓

6691 2.49.1 Definitions

6692 This section defines the key type "CKK_ARIA" for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
 6693 attribute of key objects.

6694 Mechanisms:

6695 CKM_ARIA_KEY_GEN
 6696 CKM_ARIA_ECB
 6697 CKM_ARIA_CBC
 6698 CKM_ARIA_MAC
 6699 CKM_ARIA_MAC_GENERAL
 6700 CKM_ARIA_CBC_PAD

6701 2.49.2 Aria secret key objects

6702 ARIA secret key objects (object class CKO_SECRET_KEY, key type CKK_ARIA) hold ARIA keys. The
 6703 following table defines the ARIA secret key object attributes, in addition to the common attributes defined
 6704 for this object class:

6705 *Table 185, ARIA Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (16, 24, or 32 bytes)
CKA_VALUE_LEN ^{2,3,6}	CK ULONG	Length in bytes of key value

6706 - Refer to [PKCS11-Base] table 11 for footnotes.

6707 The following is a sample template for creating an ARIA secret key object:

```
6708 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
6709 CK_KEY_TYPE keyType = CKK_ARIA;
6710 CK_UTF8CHAR label[] = "An ARIA secret key object";
6711 CK_BYTE value[] = { ... };
6712 CK_BBOOL true = CK_TRUE;
6713 CK_ATTRIBUTE template[] = {
6714     {CKA_CLASS, &class, sizeof(class)},
6715     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6716     {CKA_TOKEN, &true, sizeof(true)},
6717     {CKA_LABEL, label, sizeof(label)-1},
6718     {CKA_ENCRYPT, &true, sizeof(true)},
```

```
6719     { CKA_VALUE, value, sizeof(value) }  
6720 } ;
```

6721 **2.49.3 ARIA key generation**

6722 The ARIA key generation mechanism, denoted CKM_ARIA_KEY_GEN, is a key generation mechanism
6723 for Aria.

6724 It does not have a parameter.

6725 The mechanism generates ARIA keys with a particular length in bytes, as specified in the
6726 **CKA_VALUE_LEN** attribute of the template for the key.

6727 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
6728 key. Other attributes supported by the ARIA key type (specifically, the flags indicating which functions the
6729 key supports) may be specified in the template for the key, or else are assigned default initial values.

6730 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6731 specify the supported range of ARIA key sizes, in bytes.

6732 **2.49.4 ARIA-ECB**

6733 ARIA-ECB, denoted **CKM_ARIA_ECB**, is a mechanism for single- and multiple-part encryption and
6734 decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

6735 It does not have a parameter.

6736 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
6737 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
6738 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
6739 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
6740 length as the padded input data. It does not wrap the key type, key length, or any other information about
6741 the key; the application must convey these separately.

6742 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
6743 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
6744 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
6745 attribute of the new key; other attributes required by the key type must be specified in the template.

6746 Constraints on key types and the length of data are summarized in the following table:

6747 *Table 186, ARIA-ECB: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of block size	
C_UnwrapKey	CKK_ARIA	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

6748 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6749 specify the supported range of ARIA key sizes, in bytes.

6750 **2.49.5 ARIA-CBC**

6751 ARIA-CBC, denoted **CKM_ARIA_CBC**, is a mechanism for single- and multiple-part encryption and
6752 decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

6753 It has a parameter, a 16-byte initialization vector.

6754 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
 6755 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
 6756 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
 6757 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
 6758 length as the padded input data. It does not wrap the key type, key length, or any other information about
 6759 the key; the application must convey these separately.

6760 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
 6761 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
 6762 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
 6763 attribute of the new key; other attributes required by the key type must be specified in the template.

6764 Constraints on key types and the length of data are summarized in the following table:

6765 *Table 187, ARIA-CBC: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_Decrypt	CKK_ARIA	multiple of block size	same as input length	no final part
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of the block size	
C_UnwrapKey	CKK_ARIA	multiple of block size	determined by type of key being unwrapped or CKA_VALUE_LEN	

6766 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
 6767 specify the supported range of Aria key sizes, in bytes.

6768 2.49.6 ARIA-CBC with PKCS padding

6769 ARIA-CBC with PKCS padding, denoted **CKM_ARIA_CBC_PAD**, is a mechanism for single- and
 6770 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block
 6771 chaining mode; and the block cipher padding method detailed in PKCS #7.

6772 It has a parameter, a 16-byte initialization vector.

6773 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
 6774 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified
 6775 for the **CKA_VALUE_LEN** attribute.

6776 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
 6777 Diffie-Hellman, X9.42 Diffie-Hellman, EC (also related to ECDSA) and DSA private keys (see Section
 6778 TBA for details). The entries in the table below for data length constraints when wrapping and
 6779 unwrapping keys do not apply to wrapping and unwrapping private keys.

6780 Constraints on key types and the length of data are summarized in the following table:

6781 Table 188, ARIA-CBC with PKCS Padding: Key and Data Length

Function	Key type	Input length	Output length
C_Encrypt	CKK_ARIA	any	input length rounded up to multiple of the block size
C_Decrypt	CKK_ARIA	multiple of block size	between 1 and block size bytes shorter than input length
C_WrapKey	CKK_ARIA	any	input length rounded up to multiple of the block size
C_UnwrapKey	CKK_ARIA	multiple of block size	between 1 and block length bytes shorter than input length

6782 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6783 specify the supported range of ARIA key sizes, in bytes.

6784 2.49.7 General-length ARIA-MAC

6785 General-length ARIA -MAC, denoted **CKM_ARIA_MAC_GENERAL**, is a mechanism for single- and
6786 multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

6787 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
6788 desired from the mechanism.

6789 The output bytes from this mechanism are taken from the start of the final ARIA cipher block produced in
6790 the MACing process.

6791 Constraints on key types and the length of data are summarized in the following table:

6792 Table 189, General-length ARIA-MAC: Key and Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_ARIA	any	1-block size, as specified in parameters
C_Verify	CKK_ARIA	any	1-block size, as specified in parameters

6793 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6794 specify the supported range of ARIA key sizes, in bytes.

6795 2.49.8 ARIA-MAC

6796 ARIA-MAC, denoted by **CKM_ARIA_MAC**, is a special case of the general-length ARIA-MAC
6797 mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

6798 It does not have a parameter.

6799 Constraints on key types and the length of data are summarized in the following table:

6800 Table 190, ARIA-MAC: Key and Data Length

Function	Key type	Data length	Signature length
C_Sign	CKK_ARIA	any	½ block size (8 bytes)
C_Verify	CKK_ARIA	any	½ block size (8 bytes)

6801 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6802 specify the supported range of ARIA key sizes, in bytes.

6803 2.50 Key derivation by data encryption - ARIA

6804 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
6805 They are for use with the C_DeriveKey function.

6806 **2.50.1 Definitions**

6807 Mechanisms:

```
6808     CKM_ARIA_ECB_ENCRYPT_DATA
6809     CKM_ARIA_CBC_ENCRYPT_DATA
6810
6811     typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {
6812         CK_BYTE        iv[16];
6813         CK_BYTE_PTR   pData;
6814         CK ULONG      length;
6815     } CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;
6816
6817     typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR
6818             CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6819 **2.50.2 Mechanism Parameters**

6820 Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

6821 *Table 191, Mechanism Parameters for Aria-based key derivation*

CKM_ARIA_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_ARIA_CBC_ENCRYPT_DATA	Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6822

6823 **2.51 SEED**

6824 SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It
6825 has a 128-bit key size and a 128-bit block size.

6826 Its specification has been published as Internet [RFC 4269].

6827 RFCs have been published defining the use of SEED in

6828 TLS <ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt>

6829 IPsec <ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt>

6830 CMS <ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt>

6831

6832 TLS cipher suites that use SEED include:

```
6833     CipherSuite TLS_RSA_WITH_SEED_CBC_SHA      = { 0x00,
6834         0x96};
6835     CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA    = { 0x00,
6836         0x97};
6837     CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA    = { 0x00,
6838         0x98};
6839     CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA   = { 0x00,
6840         0x99};
```

```

6841     CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00,
6842         0x9A};
6843     CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00,
6844         0x9B};

```

6845

6846 As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as
6847 in a MAC algorithm such as HMAC.

6848 OIDs have been published for all these uses. A list may be seen at
6849 <http://www.alvestrand.no/objectid/1.2.410.200004.1.html>

6850

6851 *Table 192, SEED Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SEED_KEY_GEN					✓		
CKM_SEED_ECB			✓				
CKM_SEED_CBC			✓				
CKM_SEED_CBC_PAD	✓					✓	
CKM_SEED_MAC_GENERAL			✓				
CKM_SEED_MAC				✓			
CKM_SEED_ECB_ENCRYPT_DATA							✓
CKM_SEED_CBC_ENCRYPT_DATA							✓

6852 2.51.1 Definitions

6853 This section defines the key type “CKK_SEED” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE
6854 attribute of key objects.

6855 Mechanisms:

```

6856     CKM_SEED_KEY_GEN
6857     CKM_SEED_ECB
6858     CKM_SEED_CBC
6859     CKM_SEED_MAC
6860     CKM_SEED_MAC_GENERAL
6861     CKM_SEED_CBC_PAD

```

6862

6863 For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the CK_MECHANISM_INFO
6864 are always 16.

6865 2.51.2 SEED secret key objects

6866 SEED secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_SEED**) hold SEED keys.
6867 The following table defines the secret key object attributes, in addition to the common attributes defined
6868 for this object class:

6869 Table 193, SEED Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 16 bytes long)

6870 - Refer to [PKCS11-Base] table 11 for footnotes.

6871 The following is a sample template for creating a SEED secret key object:

```
6872 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
6873 CK_KEY_TYPE keyType = CKK_SEED;
6874 CK_UTF8CHAR label[] = "A SEED secret key object";
6875 CK_BYTE value[] = {...};
6876 CK_BBOOL true = CK_TRUE;
6877 CK_ATTRIBUTE template[] = {
6878     {CKA_CLASS, &class, sizeof(class)},
6879     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6880     {CKA_TOKEN, &true, sizeof(true)},
6881     {CKA_LABEL, label, sizeof(label)-1},
6882     {CKA_ENCRYPT, &true, sizeof(true)},
6883     {CKA_VALUE, value, sizeof(value)}}
6884 };
```

6885 2.51.3 SEED key generation

6886 The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism
6887 for SEED.

6888 It does not have a parameter.

6889 The mechanism generates SEED keys.

6890 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
6891 key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions
6892 the key supports) may be specified in the template for the key, or else are assigned default initial values.

6893 2.51.4 SEED-ECB

6894 SEED-ECB, denoted **CKM_SEED_ECB**, is a mechanism for single- and multiple-part encryption and
6895 decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

6896 It does not have a parameter.

6897 2.51.5 SEED-CBC

6898 SEED-CBC, denoted **CKM_SEED_CBC**, is a mechanism for single- and multiple-part encryption and
6899 decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

6900 It has a parameter, a 16-byte initialization vector.

6901 2.51.6 SEED-CBC with PKCS padding

6902 SEED-CBC with PKCS padding, denoted **CKM_SEED_CBC_PAD**, is a mechanism for single- and
6903 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-
6904 block chaining mode; and the block cipher padding method detailed in PKCS #7.

6905 It has a parameter, a 16-byte initialization vector.

6906 2.51.7 General-length SEED-MAC

6907 General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and
6908 multiple-part signatures and verification, based on SEED and data authentication as defined in 0.

6909 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
6910 desired from the mechanism.

6911 The output bytes from this mechanism are taken from the start of the final cipher block produced in the
6912 MACing process.

6913 2.51.8 SEED-MAC

6914 SEED-MAC, denoted by **CKM_SEED_MAC**, is a special case of the general-length SEED-MAC
6915 mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

6916 It does not have a parameter.

6917 2.52 Key derivation by data encryption - SEED

6918 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
6919 They are for use with the C_DeriveKey function.

6920 2.52.1 Definitions

6921 Mechanisms:

6922 CKM_SEED_ECB_ENCRYPT_DATA

6923 CKM_SEED_CBC_ENCRYPT_DATA

6925 `typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS
 CK_CBC_ENCRYPT_DATA_PARAMS;`

6928 `typedef CK_CBC_ENCRYPT_DATA_PARAMS CK_PTR
 CK_CBC_ENCRYPT_DATA_PARAMS_PTR;`

6930 2.52.2 Mechanism Parameters

6931 *Table 194, Mechanism Parameters for SEED-based key derivation*

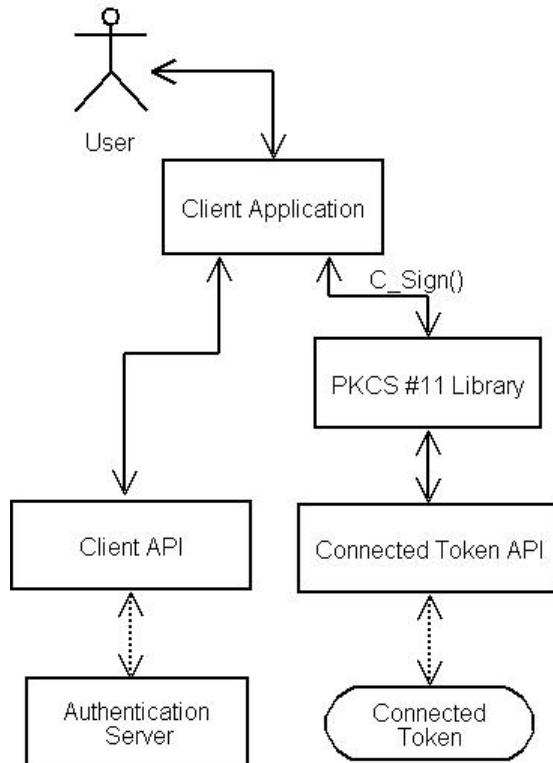
CKM_SEED_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_SEED_CBC_ENCRYPT_DATA	Uses CK_CBC_ENCRYPT_DATA_PARAMS. Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

6932 2.53 OTP

6934 2.53.1 Usage overview

6935 OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases
6936 can be categorized according to the type of sought functionality.

6937 **2.53.2 Case 1: Generation of OTP values**

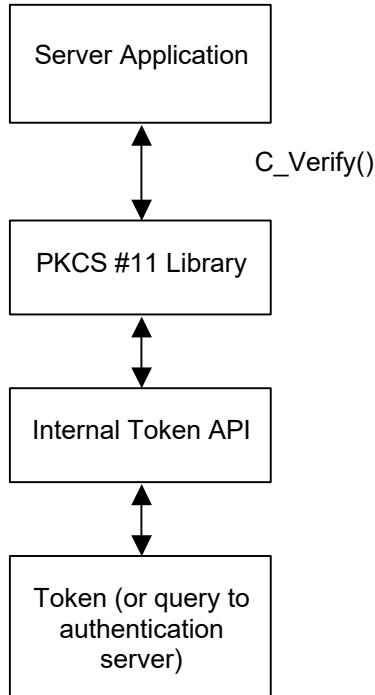


6938

6939 *Figure 1: Retrieving OTP values through C_Sign*

6940 Figure 1 shows an integration of PKCS #11 into an application that needs to authenticate users holding
6941 OTP tokens. In this particular example, a connected hardware token is used, but a software token is
6942 equally possible. The application invokes **C_Sign** to retrieve the OTP value from the token. In the
6943 example, the application then passes the retrieved OTP value to a client API that sends it via the network
6944 to an authentication server. The client API may implement a standard authentication protocol such as
6945 RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's
6946 ACE/Agent® software.

6947 **2.53.3 Case 2: Verification of provided OTP values**

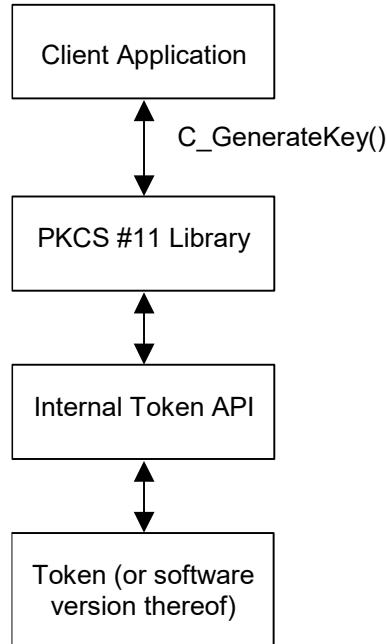


6948

6949 *Figure 2: Server-side verification of OTP values*

6950 Figure 2 illustrates the server-side equivalent of the scenario depicted in Figure 1. In this case, a server
6951 application invokes **C_Verify** with the received OTP value as the signature value to be verified.

6952 **2.53.4 Case 3: Generation of OTP keys**



6953

6954 *Figure 3: Generation of an OTP key*

6955 Figure 3 shows an integration of PKCS #11 into an application that generates OTP keys. The application
6956 invokes **C_GenerateKey** to generate an OTP key of a particular type on the token. The key may
6957 subsequently be used as a basis to generate OTP values.

6958

2.53.5 OTP objects

6959

2.53.5.1 Key objects

6960 OTP key objects (object class **CKO OTP KEY**) hold secret keys used by OTP tokens. The following
6961 table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys,
6962 all of which are inherited by this class:

6963 Table 195: Common OTP key attributes

Attribute	Data type	Meaning
CKA OTP FORMAT	CK ULONG	Format of OTP values produced with this key: CK OTP FORMAT DECIMAL = Decimal (default) (UTF8-encoded) CK OTP FORMAT HEXADECIMAL = Hexadecimal (UTF8-encoded) CK OTP FORMAT ALPHANUMERIC = Alphanumeric (UTF8-encoded) CK OTP FORMAT BINARY = Only binary values.
CKA OTP LENGTH ⁹	CK ULONG	Default length of OTP values (in the CKA OTP FORMAT) produced with this key.
CKA OTP USER FRIENDLY MODE ⁹	CK BBOOL	Set to CK TRUE when the token is capable of returning OTPs suitable for human consumption. See the description of CKF_USER_FRIENDLY OTP below.
CKA OTP CHALLENGE REQUIREMENT ⁹	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A challenge must be supplied. CK OTP PARAM OPTIONAL = A challenge may be supplied but need not be. CK OTP PARAM IGNORED = A challenge, if supplied, will be ignored.
CKA OTP TIME REQUIREMENT ⁹	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A time value must be supplied. CK OTP PARAM OPTIONAL = A time value may be supplied but need not be. CK OTP PARAM IGNORED = A time value, if supplied, will be ignored.

CKA OTP COUNTER REQUIREMENT ⁹	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A counter value must be supplied. CK OTP PARAM OPTIONAL = A counter value may be supplied but need not be. CK OTP PARAM IGNORED = A counter value, if supplied, will be ignored.
CKA OTP PIN REQUIREMENT ⁹	CK ULONG	Parameter requirements when generating or verifying OTP values with this key: CK OTP PARAM MANDATORY = A PIN value must be supplied. CK OTP PARAM OPTIONAL = A PIN value may be supplied but need not be (if not supplied, then library will be responsible for collecting it) CK OTP PARAM IGNORED = A PIN value, if supplied, will be ignored.
CKA OTP COUNTER	Byte array	Value of the associated internal counter. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP TIME	RFC 2279 string	Value of the associated internal UTC time in the form YYYYMMDDhhmmss. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP USER IDENTIFIER	RFC 2279 string	Text string that identifies a user associated with the OTP key (may be used to enhance the user experience). Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP SERVICE IDENTIFIER	RFC 2279 string	Text string that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP SERVICE LOGO	Byte array	Logotype image that identifies a service that may validate OTPs generated by this key. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA OTP SERVICE LOGO TYPE	RFC 2279 string	MIME type of the CKA OTP SERVICE LOGO attribute value. Default value is empty (i.e. <i>ulValueLen</i> = 0).
CKA VALUE ^{1, 4, 6, 7}	Byte array	Value of the key.
CKA VALUE LEN ^{2, 3}	CK ULONG	Length in bytes of key value.

6965 Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-
 6966 PKCS #11 application should therefore always consult the state of the CKA OTP_PIN_REQUIREMENT
 6967 attribute before each call to **C_SignInit**, as the value of this attribute may change dynamically.
 6968 For OTP tokens with multiple keys, the keys may be enumerated using **C_FindObjects**. The
 6969 **CKA OTP SERVICE IDENTIFIER** and/or the **CKA OTP SERVICE LOGO** attribute may be used to
 6970 distinguish between keys. The actual choice of key for a particular operation is however application-
 6971 specific and beyond the scope of this document.
 6972 For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set as required.

6973 **2.53.6 OTP-related notifications**

6974 This document extends the set of defined notifications as follows:

6975 CKN OTP CHANGED	<i>Cryptoki is informing the application that the OTP for a key on a 6976 connected token just changed. This notification is particularly useful 6977 when applications wish to display the current OTP value for time- 6978 based mechanisms.</i>
-----------------------------	---

6979 **2.53.7 OTP mechanisms**

6980 The following table shows, for the OTP mechanisms defined in this document, their support by different
 6981 cryptographic operations. For any particular token, of course, a particular operation may well support
 6982 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
 6983 mechanism for some operation supports any other mechanism for any other operation (or even supports
 6984 that same mechanism for any other operation).

6985 *Table 196: OTP mechanisms vs. applicable functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SECURID_KEY_GEN					✓		
CKM_SECURID		✓					
CKM_HOTP_KEY_GEN					✓		
CKM_HOTP		✓					
CKM_ACTI_KEY_GEN					✓		
CKM_ACTI		✓					

6986 The remainder of this section will present in detail the OTP mechanisms and the parameters that are
 6987 supplied to them.

6988 **2.53.7.1 OTP mechanism parameters**

6989 ◆ **CK_PARAM_TYPE**

6990 **CK_PARAM_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

6991 **typedef CK ULONG CK_PARAM_TYPE;**

6992 The following **CK_PARAM_TYPE** types are defined:

6993 *Table 197, OTP parameter types*

Parameter	Data type	Meaning
CK OTP PIN	RFC 2279 string	A UTF8 string containing a PIN for use when computing or verifying PIN-based OTP values.
CK OTP CHALLENGE	Byte array	Challenge to use when computing or verifying challenge-based OTP values.
CK OTP TIME	RFC 2279 string	UTC time value in the form YYYYMMDDhhmmss to use when computing or verifying time-based OTP values.
CK OTP COUNTER	Byte array	Counter value to use when computing or verifying counter-based OTP values.
CK OTP FLAGS	CK_FLAGS	Bit flags indicating the characteristics of the sought OTP as defined below.
CK OTP OUTPUT LENGTH	CK ULONG	Desired output length (overrides any default value). A Cryptoki library will return CKR_MECHANISM_PARAM_INVALID if a provided length value is not supported.
CK OTP FORMAT	CK ULONG	Returned OTP format (allowed values are the same as for CKA OTP FORMAT). This parameter is only intended for C_Sign output, see paragraphs below. When not present, the returned OTP format will be the same as the value of the CKA OTP FORMAT attribute for the key in question.
CK OTP VALUE	Byte array	An actual OTP value. This parameter type is intended for C_Sign output, see paragraphs below.

6994

6995 The following table defines the possible values for the CK OTP FLAGS type:

6996 *Table 198: OTP Mechanism Flags*

Bit flag	Mask	Meaning
CKF_NEXT OTP	0x00000001	True (i.e. set) if the OTP computation shall be for the next OTP, rather than the current one (current being interpreted in the context of the algorithm, e.g. for the current counter value or current time window). A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if the CKF_NEXT OTP flag is set and the OTP mechanism in question does not support the concept of “next” OTP or the library is not capable of generating the next OTP ⁹ .

9 Applications that may need to retrieve the next OTP should be prepared to handle this situation. For example, an application could store the OTP value returned by **C_Sign** so that, if a next OTP is required, it can compare it to the OTP value returned by subsequent calls to **C_Sign** should it turn out that the library does not support the CKF_NEXT OTP flag.

Bit flag	Mask	Meaning
CKF_EXCLUDE_TIME	0x00000002	True (i.e. set) if the OTP computation must not include a time value. Will have an effect only on mechanisms that do include a time value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_COUNTER	0x00000004	True (i.e. set) if the OTP computation must not include a counter value. Will have an effect only on mechanisms that do include a counter value in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_CHALLENGE	0x00000008	True (i.e. set) if the OTP computation must not include a challenge. Will have an effect only on mechanisms that do include a challenge in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_EXCLUDE_PIN	0x00000010	True (i.e. set) if the OTP computation must not include a PIN value. Will have an effect only on mechanisms that do include a PIN in the OTP computation and then only if the mechanism (and token) allows exclusion of this value. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if exclusion of the value is not allowed.
CKF_USER_FRIENDLY OTP	0x00000020	True (i.e. set) if the OTP returned shall be in a form suitable for human consumption. If this flag is set, and the call is successful, then the returned CK_OTP_VALUE shall be a UTF8-encoded printable string. A Cryptoki library shall return CKR_MECHANISM_PARAM_INVALID if this flag is set when CKA OTP USER FRIENDLY MODE for the key in question is CK FALSE.

6997 Note: Even if CKA OTP FORMAT is not set to CK OTP FORMAT BINARY, then there may still be
 6998 value in setting the CKF_USER_FRIENDLY OTP flag (assuming CKA OTP USER FRIENDLY MODE
 6999 is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may
 7000 become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned
 7001 OTP value to a user should not set the CKF_USER_FRIENDLY OTP flag.

◆ **CK OTP PARAM; CK OTP PARAM PTR**

7002 **CK OTP PARAM** is a structure that includes the type, value, and length of an OTP parameter. It is
 7003 defined as follows:

```

7005     typedef struct CK OTP PARAM {
7006         CK PARAM TYPE type;
7007         CK VOID PTR pValue;
7008         CK ULONG ulValueLen;
7009     } CK OTP PARAM;
7010
7011     The fields of the structure have the following meanings:
7012         type      the parameter type
7013         pValue    pointer to the value of the parameter
7014         ulValueLen length in bytes of the value
7015
7016     If a parameter has no value, then ulValueLen = 0, and the value of pValue is irrelevant. Note that pValue is a "void" pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (i.e., without word-alignment errors).
7017     CK OTP PARAM PTR is a pointer to a CK OTP PARAM.
7018
7019     ♦ CK OTP PARAMS; CK OTP PARAMS PTR
7020     CK OTP PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:
7021
7022     typedef struct CK OTP PARAMS {
7023         CK OTP PARAM PTR pParams;
7024         CK ULONG ulCount;
7025     } CK OTP PARAMS;
7026
7027     The fields of the structure have the following meanings:
7028         pParams    pointer to an array of OTP parameters
7029         ulCount   the number of parameters in the array
7030
7031     CK OTP PARAMS PTR is a pointer to a CK OTP PARAMS.
7032
7033     When calling C_SignInit or C_VerifyInit with a mechanism that takes a CK OTP PARAMS structure as a parameter, the CK OTP PARAMS structure shall be populated in accordance with the CKA OTP X REQUIREMENT key attributes for the identified key, where X is PIN, CHALLENGE, TIME, or COUNTER.
7034
7035     For example, if CKA OTP TIME REQUIREMENT = CK OTP PARAM MANDATORY, then the CK OTP TIME parameter shall be present. If CKA OTP TIME REQUIREMENT = CK OTP PARAM OPTIONAL, then a CK OTP TIME parameter may be present. If it is not present, then the library may collect it (during the C_Sign call). If CKA OTP TIME REQUIREMENT = CK OTP PARAM IGNORED, then a provided CK OTP TIME parameter will always be ignored.
7036     Additionally, a provided CK OTP TIME parameter will always be ignored if CKF EXCLUDE TIME is set in a CK OTP FLAGS parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (CKR MECHANISM PARAM INVALID) to set the CKF EXCLUDE TIME flag when the CKA OTP TIME REQUIREMENT attribute is CK OTP PARAM MANDATORY.
7037
7038     The above discussion holds for all CKA OTP X REQUIREMENT attributes (i.e.,
7039     CKA OTP PIN REQUIREMENT, CKA OTP CHALLENGE REQUIREMENT,
7040     CKA OTP COUNTER REQUIREMENT, CKA OTP TIME REQUIREMENT). A library may set a particular CKA OTP X REQUIREMENT attribute to CK OTP PARAM OPTIONAL even if it is required

```

7049 by the mechanism as long as the token (or the library itself) has the capability of providing the value to the
7050 computation. One example of this is a token with an on-board clock.

7051 In addition, applications may use the CK OTP FLAGS, the CK OTP FORMAT and the
7052 CKA OTP LENGTH parameters to set additional parameters.

7053

7054 ◆ **CK OTP SIGNATURE INFO, CK OTP SIGNATURE INFO PTR**

7055 **CK OTP SIGNATURE INFO** is a structure that is returned by all OTP mechanisms in successful calls to
7056 **C_Sign** (**C_SignFinal**). The structure informs applications of actual parameter values used in particular
7057 OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key
7058 belongs to the class **CKO OTP KEY** and is defined as follows:

```
7059     typedef struct CK OTP SIGNATURE INFO {  
7060         CK OTP PARAM PTR pParams;  
7061         CK ULONG ulCount;  
7062     } CK OTP SIGNATURE INFO;
```

7063 The fields of the structure have the following meanings:

7064 *pParams* pointer to an array of OTP parameter values

7065 *ulCount* the number of parameters in the array

7066 After successful calls to **C_Sign** or **C_SignFinal** with an OTP mechanism, the *pSignature* parameter will
7067 be set to point to a **CK OTP SIGNATURE INFO** structure. One of the parameters in this structure will be
7068 the OTP value itself, identified with the **CK OTP VALUE** tag. Other parameters may be present for
7069 informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP
7070 validations, authentication protocols may permit authenticating parties to send some or all of these
7071 parameters in addition to OTP values themselves. Applications should therefore check for their presence
7072 in returned **CK OTP SIGNATURE INFO** values whenever such circumstances apply.

7073 Since **C_Sign** and **C_SignFinal** follows the convention described in [PKCS11-Base] Section 5.2 on
7074 producing output, a call to **C_Sign** (or **C_SignFinal**) with *pSignature* set to **NULL_PTR** will return (in the
7075 *pulSignatureLen* parameter) the required number of bytes to hold the **CK OTP SIGNATURE INFO**
7076 structure as well as all the data in all its **CK OTP PARAM** components. If an application allocates a
7077 memory block based on this information, it shall therefore not subsequently de-allocate components of
7078 such a received value but rather de-allocate the complete **CK OTP PARAMS** structure itself. A Cryptoki
7079 library that is called with a non-**NULL** *pSignature* pointer will assume that it points to a *contiguous*
7080 memory block of the size indicated by the *pulSignatureLen* parameter.

7081 When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself,
7082 e.g. the value of the **CK OTP VALUE** component of a **CK OTP PARAM** structure returned by a call to
7083 **C_Sign**. The **CK OTP PARAM** value supplied in the **C_VerifyInit** call sets the values to use in the
7084 verification operation.

7085 **CK OTP SIGNATURE INFO PTR** points to a **CK OTP SIGNATURE INFO**.

7086 **2.53.8 RSA SecurID**

7087 **2.53.8.1 RSA SecurID secret key objects**

7088 RSA SecurID secret key objects (object class **CKO OTP KEY**, key type **CKK SECURID**) hold RSA
7089 SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in
7090 addition to the common attributes defined for this object class:

7091 Table 199, RSA SecurID secret key object attributes

Attribute	Data type	Meaning
CKA OTP TIME INTERVAL ¹	CK ULONG	Interval between OTP values produced with this key, in seconds. Default is 60.

7092 Refer to [PKCS11-Base] table 11 for footnotes.

7093 The following is a sample template for creating an RSA SecurID secret key object:

```
7094 CK_OBJECT_CLASS class = CKO OTP KEY;
7095 CK_KEY_TYPE keyType = CKK SECURID;
7096 CK_DATE endDate = {...};
7097 CK_UTF8CHAR label[] = "RSA SecurID secret key object";
7098 CK_BYTE keyId[] = {...};
7099 CK ULONG outputFormat = CK OTP FORMAT DECIMAL;
7100 CK ULONG outputLength = 6;
7101 CK ULONG needPIN = CK OTP PARAM MANDATORY;
7102 CK ULONG timeInterval = 60;
7103 CK_BYTE value[] = {...};
7104     CK_BBOOL true = CK TRUE;
7105 CK_ATTRIBUTE template[] = {
7106     {CKA CLASS, &class, sizeof(class)},
7107     {CKA KEY TYPE, &keyType, sizeof(keyType)},
7108     {CKA END DATE, &endDate, sizeof(endDate)},
7109     {CKA TOKEN, &true, sizeof(true)},
7110     {CKA SENSITIVE, &true, sizeof(true)},
7111     {CKA LABEL, label, sizeof(label)-1},
7112     {CKA SIGN, &true, sizeof(true)},
7113     {CKA VERIFY, &true, sizeof(true)},
7114     {CKA ID, keyId, sizeof(keyId)},
7115     {CKA OTP FORMAT, &outputFormat, sizeof(outputFormat)},
7116     {CKA OTP LENGTH, &outputLength, sizeof(outputLength)},
7117     {CKA OTP PIN REQUIREMENT, &needPIN, sizeof(needPIN)},
7118     {CKA OTP TIME INTERVAL, &timeInterval,
7119         sizeof(timeInterval)},
7120     {CKA VALUE, value, sizeof(value)}}
7121 };
```

7122 2.53.8.2 RSA SecurID key generation

7123 The RSA SecurID key generation mechanism, denoted **CKM SECURID KEY GEN**, is a key generation
7124 mechanism for the RSA SecurID algorithm.

7125 It does not have a parameter.

7126 The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the
7127 template for the key.

7128 The mechanism contributes at least the **CKA CLASS**, **CKA KEY TYPE**, **CKA VALUE LEN**, and
7129 **CKA VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be
7130 specified in the template for the key, or else are assigned default initial values.

7131 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK MECHANISM INFO** structure
7132 specify the supported range of SecurID key sizes, in bytes.

2.53.8.3 SecurID OTP generation and validation

CKM_SECURID is the mechanism for the retrieval and verification of RSA SecurID OTP values.

The mechanism takes a pointer to a **CK OTP_PARAMS** structure as a parameter.

When signing or verifying using the **CKM_SECURID** mechanism, *pData* shall be set to **NULL_PTR** and *ulDataLen* shall be set to 0.

2.53.8.4 Return values

Support for the **CKM_SECURID** mechanism extends the set of return values for **C_Verify** with the following values:

- **CKR_NEW_PIN_MODE**: The supplied OTP was not accepted and the library requests a new OTP computed using a new PIN. The new PIN is set through means out of scope for this document.
- **CKR_NEXT OTP**: The supplied OTP was correct but indicated a larger than normal drift in the token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the application should provide the next one-time password to the library for verification.

2.53.9 OATH HOTP

2.53.9.1 OATH HOTP secret key objects

HOTP secret key objects (object class **CKO OTP KEY**, key type **CKK_HOTP**) hold generic secret keys and associated counter values.

The **CKA OTP COUNTER** value may be set at key generation; however, some tokens may set it to a fixed initial value. Depending on the token's security policy, this value may not be modified and/or may not be revealed if the object has its **CKA SENSITIVE** attribute set to **CK_TRUE** or its **CKA_EXTRACTABLE** attribute set to **CK_FALSE**.

For HOTP keys, the **CKA OTP COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e. network byte order) form. The same holds true for a **CK OTP COUNTER** value in a **CK OTP PARAM** structure.

The following is a sample template for creating a HOTP secret key object:

```
7158     CK_OBJECT_CLASS class = CKO OTP KEY;
7159     CK_KEY_TYPE keyType = CKK_HOTP;
7160     CK_UTF8CHAR label[] = "HOTP secret key object";
7161     CK_BYTE keyId[] = {...};
7162     CK_ULONG outputFormat = CK OTP FORMAT DECIMAL;
7163     CK_ULONG outputLength = 6;
7164     CK_DATE endDate = {...};
7165     CK_BYTE counterValue[8] = {0};
7166     CK_BYTE value[] = {...};
7167     CK_BBOOL true = CK TRUE;
7168     CK_ATTRIBUTE template[] = {
7169         {CKA CLASS, &class, sizeof(class)},
7170         {CKA KEY TYPE, &keyType, sizeof(keyType)},
7171         {CKA END DATE, &endDate, sizeof(endDate)},
7172         {CKA TOKEN, &true, sizeof(true)},
7173         {CKA SENSITIVE, &true, sizeof(true)},
7174         {CKA LABEL, label, sizeof(label)-1},
7175         {CKA SIGN, &true, sizeof(true)},
7176         {CKA VERIFY, &true, sizeof(true)},
7177         {CKA ID, keyId, sizeof(keyId)},
```

```
7178     { CKA OTP FORMAT, &outputFormat, sizeof(outputFormat) },
7179     { CKA OTP LENGTH, &outputLength, sizeof(outputLength) },
7180     { CKA OTP COUNTER, counterValue, sizeof(counterValue) },
7181     { CKA VALUE, value, sizeof(value) }
7182 };
```

7183 2.53.9.2 HOTP key generation

7184 The HOTP key generation mechanism, denoted **CKM_HOTP_KEY_GEN**, is a key generation mechanism
7185 for the HOTP algorithm.

7186 It does not have a parameter.

7187 The mechanism generates HOTP keys with a particular set of attributes as specified in the template for
7188 the key.

7189 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA OTP COUNTER**,
7190 **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the HOTP
7191 key type may be specified in the template for the key, or else are assigned default initial values.

7192 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7193 specify the supported range of HOTP key sizes, in bytes.

7194 2.53.9.3 HOTP OTP generation and validation

7195 **CKM_HOTP** is the mechanism for the retrieval and verification of HOTP OTP values based on the current
7196 internal counter, or a provided counter.

7197 The mechanism takes a pointer to a **CK OTP PARAMS** structure as a parameter.

7198 As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism,
7199 *pData* shall be set to **NULL_PTR** and *ulDataLen* shall be set to 0.

7200 For verify operations, the counter value **CK OTP COUNTER** must be provided as a **CK OTP PARAM**
7201 parameter to **C VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, *pSignature*
7202 shall be set to the OTP value itself, e.g. the value of the **CK OTP VALUE** component of a
7203 **CK OTP PARAM** structure in the case of an earlier call to **C_Sign**.

7204 2.53.10 ActivIdentity ACTI

7205 2.53.10.1 ACTI secret key objects

7206 ACTI secret key objects (object class **CKO OTP KEY**, key type **CKK_ACTI**) hold ActivIdentity ACTI
7207 secret keys.

7208 For ACTI keys, the **CKA OTP COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e.
7209 network byte order) form. The same holds true for the **CK OTP COUNTER** value in the
7210 **CK OTP PARAM** structure.

7211 The **CKA OTP COUNTER** value may be set at key generation; however, some tokens may set it to a
7212 fixed initial value. Depending on the token's security policy, this value may not be modified and/or may
7213 not be revealed if the object has its **CKA SENSITIVE** attribute set to CK_TRUE or its
7214 **CKA EXTRACTABLE** attribute set to CK_FALSE.

7215 The **CKA OTP TIME** value may be set at key generation; however, some tokens may set it to a fixed
7216 initial value. Depending on the token's security policy, this value may not be modified and/or may not be
7217 revealed if the object has its **CKA SENSITIVE** attribute set to CK_TRUE or its **CKA EXTRACTABLE**
7218 attribute set to CK_FALSE.

7219 The following is a sample template for creating an ACTI secret key object:

```
7220     CK_OBJECT_CLASS class = CKO OTP KEY;
7221     CK KEY TYPE keyType = CKK ACTI;
7222     CK UTF8CHAR label[] = "ACTI secret key object";
```

```

7223 CK_BYTE keyId[] = { ... };
7224 CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
7225 CK_ULONG outputLength = 6;
7226 CK_DATE endDate = { ... };
7227 CK_BYTE counterValue[8] = { 0 };
7228 CK_BYTE value[] = { ... };
7229 CK_BBOOL true = CK_TRUE;
7230 CK_ATTRIBUTE template[] = {
7231     { CKA_CLASS, &class, sizeof(class) },
7232     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
7233     { CKA_END_DATE, &endDate, sizeof(endDate) },
7234     { CKA_TOKEN, &true, sizeof(true) },
7235     { CKA_SENSITIVE, &true, sizeof(true) },
7236     { CKA_LABEL, label, sizeof(label)-1 },
7237     { CKA_SIGN, &true, sizeof(true) },
7238     { CKA_VERIFY, &true, sizeof(true) },
7239     { CKA_ID, keyId, sizeof(keyId) },
7240     { CKA_OTP_FORMAT, &outputFormat,
7241         sizeof(outputFormat) },
7242     { CKA_OTP_LENGTH, &outputLength,
7243         sizeof(outputLength) },
7244     { CKA_OTP_COUNTER, counterValue,
7245         sizeof(counterValue) },
7246     { CKA_VALUE, value, sizeof(value) }
7247 };

```

7248 2.53.10.2 ACTI key generation

7249 The ACTI key generation mechanism, denoted **CKM_ACTI_KEY_GEN**, is a key generation mechanism
 7250 for the ACTI algorithm.

7251 It does not have a parameter.

7252 The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the
 7253 key.

7254 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE** and
 7255 **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be
 7256 specified in the template for the key, or else are assigned default initial values.

7257 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7258 specify the supported range of ACTI key sizes, in bytes.

7259 2.53.10.3 ACTI OTP generation and validation

7260 **CKM_ACTI** is the mechanism for the retrieval and verification of ACTI OTP values.

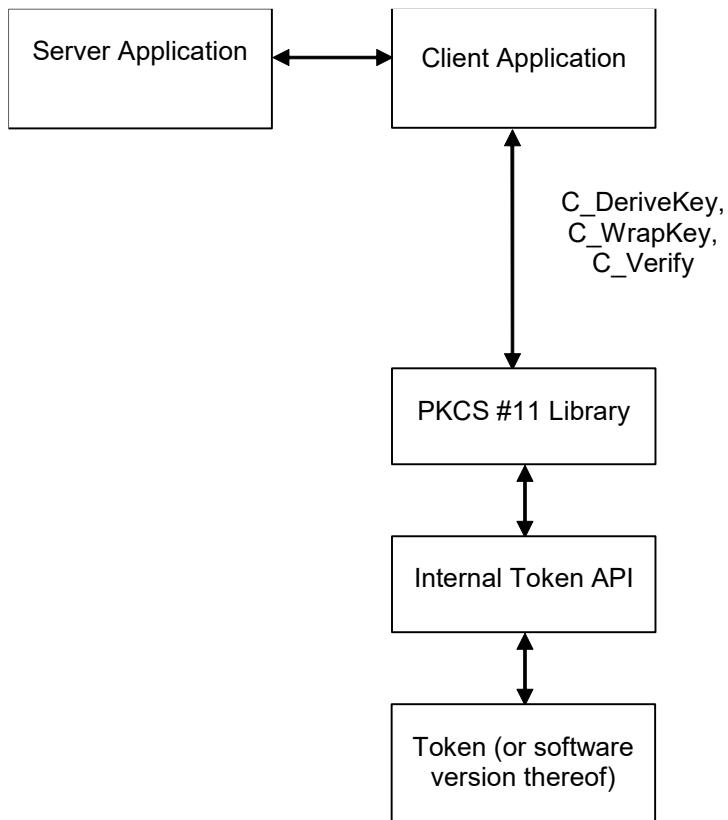
7261 The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

7262 When signing or verifying using the **CKM_ACTI** mechanism, *pData* shall be set to **NULL_PTR** and
 7263 *ulDataLen* shall be set to 0.

7264 When verifying an OTP value using the **CKM_ACTI** mechanism, *pSignature* shall be set to the OTP value
 7265 itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure in the case of
 7266 an earlier call to **C_Sign**.

7267 **2.54 CT-KIP**

7268 **2.54.1 Principles of Operation**



7269

7270 *Figure 4: PKCS #11 and CT-KIP integration*

7271 Figure 4 shows an integration of PKCS #11 into an application that generates cryptographic keys through
7272 the use of CT-KIP. The application invokes **C_DeriveKey** to derive a key of a particular type on the token.
7273 The key may subsequently be used as a basis to e.g., generate one-time password values. The
7274 application communicates with a CT-KIP server that participates in the key derivation and stores a copy
7275 of the key in its database. The key is transferred to the server in wrapped form, after a call to
7276 **C_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls
7277 to **C_Verify**.

7278 **2.54.2 Mechanisms**

7279 The following table shows, for the mechanisms defined in this document, their support by different
7280 cryptographic operations. For any particular token, of course, a particular operation may well support
7281 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
7282 mechanism for some operation supports any other mechanism for any other operation (or even supports
7283 that same mechanism for any other operation).

7284 Table 200: CT-KIP Mechanisms vs. applicable functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_KIP_DERIVE							✓
CKM_KIP_WRAP						✓	
CKM_KIP_MAC		✓					

7285 The remainder of this section will present in detail the mechanisms and the parameters that are supplied
 7286 to them.

7287 2.54.3 Definitions

7288 Mechanisms:

7289 CKM_KIP_DERIVE
 7290 CKM_KIP_WRAP
 7291 CKM_KIP_MAC

7292 2.54.4 CT-KIP Mechanism parameters

7293 ◆ CK_KIP_PARAMS; CK_KIP_PARAMS_PTR

7294 CK_KIP_PARAMS is a structure that provides the parameters to all the CT-KIP related mechanisms: The
 7295 CKM_KIP_DERIVE key derivation mechanism, the CKM_KIP_WRAP key wrap and key unwrap
 7296 mechanism, and the CKM_KIP_MAC signature mechanism. The structure is defined as follows:

```
7297     typedef struct CK_KIP_PARAMS {
7298         CK_MECHANISM_PTR pMechanism;
7299         CK_OBJECT_HANDLE hKey;
7300         CK_BYTE_PTR       pSeed;
7301         CK ULONG          ulSeedLen;
7302     } CK_KIP_PARAMS;
```

7303 The fields of the structure have the following meanings:

7304 pMechanism pointer to the underlying cryptographic mechanism (e.g. AES, SHA-
 7305 256), see further 0, Appendix D

7306 hKey handle to a key that will contribute to the entropy of the derived key
 7307 (CKM_KIP_DERIVE) or will be used in the MAC operation
 7308 (CKM_KIP_MAC)

7309 pSeed pointer to an input seed

7310 ulSeedLen length in bytes of the input seed

7311 CK_KIP_PARAMS_PTR is a pointer to a CK_KIP_PARAMS structure.

7312 2.54.5 CT-KIP key derivation

7313 The CT-KIP key derivation mechanism, denoted CKM_KIP_DERIVE, is a key derivation mechanism that
 7314 is capable of generating secret keys of potentially any type, subject to token limitations.

7315 It takes a parameter of type **CK_KIP_PARAMS** which allows for the passing of the desired underlying
 7316 cryptographic mechanism as well as some other data. In particular, when the *hKey* parameter is a handle
 7317 to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C_DeriveKey**.
 7318 The *pSeed* parameter may be used to seed the key derivation operation.
 7319 The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the
 7320 template for the key.
 7321 The mechanism contributes the **CKA_CLASS** and **CKA_VALUE** attributes to the new key. Other
 7322 attributes supported by the key type may be specified in the template for the key, or else will be assigned
 7323 default initial values. Since the mechanism is generic, the **CKA_KEY_TYPE** attribute should be set in the
 7324 template, if the key is to be used with a particular mechanism.

7325 **2.54.6 CT-KIP key wrap and key unwrap**

7326 The CT-KIP key wrap and unwrap mechanism, denoted **CKM_KIP_WRAP**, is a key wrap mechanism that
 7327 is capable of wrapping and unwrapping generic secret keys.
 7328 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
 7329 cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of
 7330 **CK_KIP_PARAMS**.

7331 **2.54.7 CT-KIP signature generation**

7332 The CT-KIP signature (MAC) mechanism, denoted **CKM_KIP_MAC**, is a mechanism used to produce a
 7333 message authentication code of arbitrary length. The keys it uses are secret keys.
 7334 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
 7335 cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed*
 7336 and the *uSeedLen* parameters of **CT_KIP_PARAMS**.
 7337 This mechanism produces a MAC of the length specified by *pulSignatureLen* parameter in calls to
 7338 **C_Sign**.
 7339 If a call to **C_Sign** with this mechanism fails, then no output will be generated.

7340 **2.55 GOST 28147-89**

7341 GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.
 7342
 7343 *Table 201, GOST 28147-89 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOST28147_KEY_GEN					✓		
CKM_GOST28147_ECB	✓					✓	
CKM_GOST28147	✓					✓	
CKM_GOST28147_MAC		✓					
CKM_GOST28147_KEY_WRAP						✓	

7344

7345 2.55.1 Definitions

7346 This section defines the key type “CKK_GOST28147” for type CK_KEY_TYPE as used in the
7347 CKA_KEY_TYPE attribute of key objects and domain parameter objects.

7348 Mechanisms:

7349 CKM_GOST28147_KEY_GEN
7350 CKM_GOST28147_ECB
7351 CKM_GOST28147
7352 CKM_GOST28147_MAC
7353 CKM_GOST28147_KEY_WRAP

7354 2.55.2 GOST 28147-89 secret key objects

7355 GOST 28147-89 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_GOST28147**) hold
7356 GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in
7357 addition to the common attributes defined for this object class:

7358 *Table 202, GOST 28147-89 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	32 bytes in little endian order
CKA_GOST28147_PARAMS ^{1,3,5}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID

7359 Refer to [PKCS11-Base] Table 11 for footnotes

7360 The following is a sample template for creating a GOST 28147-89 secret key object:

```
7361 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
7362 CK_KEY_TYPE keyType = CKK_GOST28147;
7363 CK_UTF8CHAR label[] = "A GOST 28147-89 secret key object";
7364 CK_BYTE value[32] = {...};
7365 CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,
7366 0x02, 0x1f, 0x00};
7367 CK_BBOOL true = CK_TRUE;
7368 CK_ATTRIBUTE template[] = {
7369     {CKA_CLASS, &class, sizeof(class)},
7370     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7371     {CKA_TOKEN, &true, sizeof(true)},
7372     {CKA_LABEL, label, sizeof(label)-1},
7373     {CKA_ENCRYPT, &true, sizeof(true)},
7374     {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},
7375     {CKA_VALUE, value, sizeof(value)}}
7376 };
```

7377 2.55.3 GOST 28147-89 domain parameter objects

7378 GOST 28147-89 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
7379 **CKK_GOST28147**) hold GOST 28147-89 domain parameters.

7380 The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the
7381 common attributes defined for this object class:

7382 *Table 203, GOST 28147-89 Domain Parameter Object Attributes*

Attribute	Data Type	Meaning
CKA_VALUE ¹	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.1 (type <i>Gost28147-89-ParamSetParameters</i>)
CKA_OBJECT_ID ¹	Byte array	DER-encoding of the object identifier indicating the domain parameters

7383 Refer to [PKCS11-Base] Table 11 for footnotes

7384 For any particular token, there is no guarantee that a token supports domain parameters loading up
7385 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
7386 take in account that **CKA_VALUE** attribute may be inaccessible.

7387 The following is a sample template for creating a GOST 28147-89 domain parameter object:

```
7388     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
7389     CK_KEY_TYPE keyType = CKK_GOST28147;
7390     CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic
7391         parameters object";
7392     CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
7393         0x1f, 0x00};
7394     CK_BYTE value[] = {
7395         0x30, 0x62, 0x04, 0x40, 0x4c, 0xde, 0x38, 0x9c, 0x29, 0x89, 0xef, 0xb6,
7396         0xff, 0xeb, 0x56, 0xc5, 0x5e, 0xc2, 0x9b, 0x02, 0x98, 0x75, 0x61, 0x3b,
7397         0x11, 0x3f, 0x89, 0x60, 0x03, 0x97, 0x0c, 0x79, 0x8a, 0xa1, 0xd5, 0x5d,
7398         0xe2, 0x10, 0xad, 0x43, 0x37, 0x5d, 0xb3, 0x8e, 0xb4, 0x2c, 0x77, 0xe7,
7399         0xcd, 0x46, 0xca, 0xfa, 0xd6, 0x6a, 0x20, 0x1f, 0x70, 0xf4, 0x1e, 0xa4,
7400         0xab, 0x03, 0xf2, 0x21, 0x65, 0xb8, 0x44, 0xd8, 0x02, 0x01, 0x00, 0x02,
7401         0x01, 0x40, 0x30, 0x0b, 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x0e,
7402         0x00, 0x05, 0x00
7403     };
7404     CK_BBOOL true = CK_TRUE;
7405     CK_ATTRIBUTE template[] = {
7406         {CKA_CLASS, &class, sizeof(class)},
7407         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7408         {CKA_TOKEN, &true, sizeof(true)},
7409         {CKA_LABEL, label, sizeof(label)-1},
7410         {CKA_OBJECT_ID, oid, sizeof(oid)},
7411         {CKA_VALUE, value, sizeof(value)}
7412     };

```

7413 2.55.4 GOST 28147-89 key generation

7414 The GOST 28147-89 key generation mechanism, denoted **CKM_GOST28147_KEY_GEN**, is a key
7415 generation mechanism for GOST 28147-89.

7416 It does not have a parameter.

7417 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
7418 key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object
7419 class **CKO_SECRET_KEY**.

7420 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are not
7421 used.

7422 **2.55.5 GOST 28147-89-ECB**

7423 GOST 28147-89-ECB, denoted **CKM_GOST28147_ECB**, is a mechanism for single and multiple-part
7424 encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and electronic
7425 codebook mode.

7426 It does not have a parameter.

7427 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
7428 wrap/unwrap every secret key that it supports.

7429 For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key
7430 that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple
7431 of the block size.

7432 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and truncates the result
7433 according to the **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports
7434 it, the **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the
7435 **CKA_VALUE** attribute of the new key.

7436 Constraints on key types and the length of data are summarized in the following table:

7437 *Table 204, GOST 28147-89-ECB: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_GOST28147	Multiple of block size	Same as input length
C_Decrypt	CKK_GOST28147	Multiple of block size	Same as input length
C_WrapKey	CKK_GOST28147	Any	Input length rounded up to multiple of block size
C_UnwrapKey	CKK_GOST28147	Multiple of block size	Determined by type of key being unwrapped

7438

7439 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7440 are not used.

7441 **2.55.6 GOST 28147-89 encryption mode except ECB**

7442 GOST 28147-89 encryption mode except ECB, denoted **CKM_GOST28147**, is a mechanism for single
7443 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on
7444 [GOST 28147-89] and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2.
7445 Encryption's parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

7446 It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero
7447 initialization vector is used.

7448 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
7449 wrap/unwrap every secret key that it supports.

7450 For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key
7451 that is wrapped.

7452 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as
7453 the **CKA_VALUE** attribute of the new key.

7454 Constraints on key types and the length of data are summarized in the following table:

7455 *Table 205, GOST 28147-89 encryption modes except ECB: Key and Data Length*

Function	Key type	Input length	Output length
C_Encrypt	CKK_GOST28147	Any	For counter mode and CFB is the same as input length. For CBC is the same as input length padded on the trailing end with up to block size so that the resulting length is a multiple of the block size
C_Decrypt	CKK_GOST28147	Any	
C_WrapKey	CKK_GOST28147	Any	
C_UnwrapKey	CKK_GOST28147	Any	

7456

7457 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7458 are not used.

7459 **2.55.7 GOST 28147-89-MAC**

7460 GOST 28147-89-MAC, denoted **CKM_GOST28147_MAC**, is a mechanism for data integrity and
7461 authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

7462 MACing parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

7463 The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block
7464 produced in the MACing process.

7465 It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a
7466 zero initialization vector is used.

7467 Constraints on key types and the length of data are summarized in the following table:

7468 *Table 206, GOST28147-89-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_GOST28147	Any	4 bytes
C_Verify	CKK_GOST28147	Any	4 bytes

7469

7470 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7471 are not used.

7472

7473 **2.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89**

7474 GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by
7475 **CKM_GOST28147_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on
7476 GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation
7477 mechanism for GOST 28147-89.

7478 For wrapping (**C_WrapKey**), the mechanism first computes MAC from the value of the **CKA_VALUE**
7479 attribute of the key that is wrapped and then encrypts in ECB mode the value of the **CKA_VALUE**
7480 attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

7481 For unwrapping (**C_UnwrapKey**), the mechanism first decrypts in ECB mode the 32 bytes of the key that
7482 was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC

7483 has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is
 7484 disallowed. The mechanism contributes the result as the **CKA_VALUE** attribute of the unwrapped key.
 7485 It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a
 7486 zero initialization vector is used.
 7487 Constraints on key types and the length of data are summarized in the following table:
 7488 *Table 207, GOST 28147-89 keys as KEK: Key and Data Length*

Function	Key type	Input length	Output length
C_WrapKey	CKK_GOST28147	32 bytes	36 bytes
C_UnwrapKey	CKK_GOST28147	32 bytes	36 bytes

7489
 7490 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7491 are not used.
 7492

7493 **2.56 GOST R 34.11-94**

7494 GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit
 7495 message digest defined in [GOST R 34.11-94].
 7496

7497 *Table 208, GOST R 34.11-94 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOSTR3411				✓			
CKM_GOSTR3411_HMAC		✓					

7498

7499 **2.56.1 Definitions**

7500 This section defines the key type "CKK_GOSTR3411" for type CK_KEY_TYPE as used in the
 7501 CKA_KEY_TYPE attribute of domain parameter objects.

7502 Mechanisms:

7503 CKM_GOSTR3411
 7504 CKM_GOSTR3411_HMAC

7505 **2.56.2 GOST R 34.11-94 domain parameter objects**

7506 GOST R 34.11-94 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
 7507 **CKK_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

7508 The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the
 7509 common attributes defined for this object class:

7510 Table 209, GOST R 34.11-94 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE ¹	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.2 (type <i>GostR3411-94-ParamSetParameters</i>)
CKA_OBJECT_ID ¹	Byte array	DER-encoding of the object identifier indicating the domain parameters

7511 Refer to [PKCS11-Base] Table 11 for footnotes

7512 For any particular token, there is no guarantee that a token supports domain parameters loading up
 7513 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
 7514 take in account that **CKA_VALUE** attribute may be inaccessible.

7515 The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```

7516 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
7517 CK_KEY_TYPE keyType = CKK_GOSTR3411;
7518 CK_UTF8CHAR label[] = "A GOST R34.11-94 cryptographic
7519     parameters object";
7520 CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
7521     0x1e, 0x00};
7522 CK_BYTE value[] = {
7523     0x30, 0x64, 0x04, 0x40, 0x4e, 0x57, 0x64, 0xd1, 0xab, 0x8d, 0xcb, 0xbf,
7524     0x94, 0x1a, 0x7a, 0x4d, 0x2c, 0xd1, 0x10, 0x10, 0xd6, 0xa0, 0x57, 0x35,
7525     0x8d, 0x38, 0xf2, 0xf7, 0x0f, 0x49, 0xd1, 0x5a, 0xea, 0x2f, 0x8d, 0x94,
7526     0x62, 0xee, 0x43, 0x09, 0xb3, 0xf4, 0xa6, 0xa2, 0x18, 0xc6, 0x98, 0xe3,
7527     0xc1, 0x7c, 0xe5, 0x7e, 0x70, 0x6b, 0x09, 0x66, 0xf7, 0x02, 0x3c, 0x8b,
7528     0x55, 0x95, 0xbf, 0x28, 0x39, 0xb3, 0x2e, 0xcc, 0x04, 0x20, 0x00, 0x00,
7529     0x00, 0x00,
7530     0x00, 0x00,
7531     0x00, 0x00, 0x00, 0x00, 0x00, 0x00
7532 };
7533 CK_BBOOL true = CK_TRUE;
7534 CK_ATTRIBUTE template[] = {
7535     {CKA_CLASS, &class, sizeof(class)},
7536     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7537     {CKA_TOKEN, &true, sizeof(true)},
7538     {CKA_LABEL, label, sizeof(label)-1},
7539     {CKA_OBJECT_ID, oid, sizeof(oid)},
7540     {CKA_VALUE, value, sizeof(value)}
7541 };

```

7542 2.56.3 GOST R 34.11-94 digest

7543 GOST R 34.11-94 digest, denoted **CKM_GOSTR3411**, is a mechanism for message digesting based on
 7544 GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

7545 As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter
 7546 may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357]
 7547 (section 11.2) must be used.

7548 Constraints on the length of input and output data are summarized in the following table. For single-part
 7549 digesting, the data and the digest may begin at the same location in memory.

7550 *Table 210, GOST R 34.11-94: Data Length*

Function	Input length	Digest length
C_Digest	Any	32 bytes

7551

7552 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7553 are not used.

7554 **2.56.4 GOST R 34.11-94 HMAC**

7555 GOST R 34.11-94 HMAC mechanism, denoted **CKM_GOSTR3411_HMAC**, is a mechanism for
7556 signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash
7557 function [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key
7558 type **CKK_GENERIC_SECRET** or **CKK_GOST28147**.

7559 To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC
7560 algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

7561 As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter
7562 may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357]
7563 (section 11.2) must be used.

7564 Signatures (MACs) produced by this mechanism are of 32 bytes long.

7565 Constraints on the length of input and output data are summarized in the following table:

7566 *Table 211, GOST R 34.11-94 HMAC: Key And Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_GENERIC_SECRET or CKK_GOST28147	Any	32 byte
C_Verify	CKK_GENERIC_SECRET or CKK_GOST28147	Any	32 bytes

7567 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7568 are not used.

7569 **2.57 GOST R 34.10-2001**

7570 GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following
7571 the digital signature algorithm defined in [GOST R 34.10-2001].

7572

7573 *Table 212, GOST R34.10-2001 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOSTR3410_KEY_PAIR_GEN					✓		
CKM_GOSTR3410		✓ ¹					
CKM_GOSTR3410_WITH_GOST3411		✓					
CKM_GOSTR3410_KEY_WRAP						✓	
CKM_GOSTR3410_DERIVE							✓

7574

1 Single-part operations only

7575

7576 2.57.1 Definitions

7577 This section defines the key type "CKK_GOSTR3410" for type CK_KEY_TYPE as used in the
7578 CKA_KEY_TYPE attribute of key objects and domain parameter objects.

7579 Mechanisms:

7580 CKM_GOSTR3410_KEY_PAIR_GEN
7581 CKM_GOSTR3410
7582 CKM_GOSTR3410_WITH_GOSTR3411
7583 CKM_GOSTR3410
7584 CKM_GOSTR3410_KEY_WRAP
7585 CKM_GOSTR3410_DERIVE

7586 2.57.2 GOST R 34.10-2001 public key objects

7587 GOST R 34.10-2001 public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_GOSTR3410**)
7588 hold GOST R 34.10-2001 public keys.

7589 The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the
7590 common attributes defined for this object class:

7591 *Table 213, GOST R 34.10-2001 Public Key Object Attributes*

Attribute	Data Type	Meaning
CKA_VALUE ^{1,4}	Byte array	64 bytes for public key; 32 bytes for each coordinates X and Y of elliptic curve point P(X, Y) in little endian order
CKA_GOSTR3410_PARAMS ^{1,3}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOSTR3411_PARAMS ^{1,3,8}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS ⁸	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted

7592 Refer to [PKCS11-Base] Table 11 for footnotes

7593 The following is a sample template for creating an GOST R 34.10-2001 public key object:

```

7594 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7595 CK_KEY_TYPE keyType = CKK_GOSTR3410;
7596 CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";
7597 CK_BYTE gostR3410params_oid[] =
7598     { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00 };
7599 CK_BYTE gostR3411params_oid[] =
7600     { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00 };
7601 CK_BYTE gost28147params_oid[] =
7602     { 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00 };
7603 CK_BYTE value[64] = {...};
7604 CK_BBOOL true = CK_TRUE;
7605 CK_ATTRIBUTE template[] = {
7606     { CKA_CLASS, &class, sizeof(class) },
7607     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
7608     { CKA_TOKEN, &true, sizeof(true) },
7609     { CKA_LABEL, label, sizeof(label)-1 },
7610     { CKA_GOSTR3410_PARAMS, gostR3410params_oid,
7611         sizeof(gostR3410params_oid) },
7612     { CKA_GOSTR3411_PARAMS, gostR3411params_oid,
7613         sizeof(gostR3411params_oid) },
7614     { CKA_GOST28147_PARAMS, gost28147params_oid,
7615         sizeof(gost28147params_oid) },
7616     { CKA_VALUE, value, sizeof(value) }
7617 } ;

```

7618 2.57.3 GOST R 34.10-2001 private key objects

7619 GOST R 34.10-2001 private key objects (object class **CKO_PRIVATE_KEY**, key type
 7620 **CKK_GOSTR3410**) hold GOST R 34.10-2001 private keys.

7621 The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the
 7622 common attributes defined for this object class:

7623 *Table 214, GOST R 34.10-2001 Private Key Object Attributes*

Attribute	Data Type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	32 bytes for private key in little endian order
CKA_GOSTR3410_PARAMS ^{1,4,6}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.10-2001. When key is used the domain parameter object of key type CKK_GOSTR3410 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOSTR3411_PARAMS ^{1,4,6,8}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST R 34.11-94. When key is used the domain parameter object of key type CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS ^{4,6,8}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID. The attribute value may be omitted

7624

Refer to [PKCS11-Base] Table 11 for footnotes

7625 Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain
 7626 parameters are *not* specified in the key's template. This is because GOST R 34.10-2001 private keys are
 7627 only generated as part of an GOST R 34.10-2001 key pair, and the GOST R 34.10-2001 domain
 7628 parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

7629 The following is a sample template for creating an GOST R 34.10-2001 private key object:

```

7630     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7631     CK_KEY_TYPE keyType = CKK_GOSTR3410;
7632     CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key
7633         object";
7634     CK_BYTE subject[] = {...};
7635     CK_BYTE id[] = {123};
7636     CK_BYTE gostR3410params_oid[] =
7637         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
7638     CK_BYTE gostR3411params_oid[] =
7639         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
7640     CK_BYTE gost28147params_oid[] =
7641         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
7642     CK_BYTE value[32] = {...};
7643     CK_BBOOL true = CK_TRUE;
7644     CK_ATTRIBUTE template[] = {

```

```

7645     { CKA_CLASS, &class, sizeof(class) },
7646     { CKA_KEY_TYPE, &keyType, sizeof(keyType) },
7647     { CKA_TOKEN, &true, sizeof(true) },
7648     { CKA_LABEL, label, sizeof(label)-1 },
7649     { CKA_SUBJECT, subject, sizeof(subject) },
7650     { CKA_ID, id, sizeof(id) },
7651     { CKA_SENSITIVE, &true, sizeof(true) },
7652     { CKA_SIGN, &true, sizeof(true) },
7653     { CKA_GOSTR3410_PARAMS, gostR3410params_oid,
7654         sizeof(gostR3410params_oid) },
7655     { CKA_GOSTR3411_PARAMS, gostR3411params_oid,
7656         sizeof(gostR3411params_oid) },
7657     { CKA_GOST28147_PARAMS, gost28147params_oid,
7658         sizeof(gost28147params_oid) },
7659     { CKA_VALUE, value, sizeof(value) }
7660 };
7661

```

2.57.4 GOST R 34.10-2001 domain parameter objects

GOST R 34.10-2001 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_GOSTR3410**) hold GOST R 34.10-2001 domain parameters.

The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 215, GOST R 34.10-2001 Domain Parameter Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE ¹	Byte array	DER-encoding of the domain parameters as it was introduced in [4] section 8.4 (type <i>GostR3410-2001-ParamSetParameters</i>)
CKA_OBJECT_ID ¹	Byte array	DER-encoding of the object identifier indicating the domain parameters

Refer to [PKCS11-Base] Table 11 for footnotes

For any particular token, there is no guarantee that a token supports domain parameters loading up and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should take in account that **CKA_VALUE** attribute may be inaccessible.

The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```

7673     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
7674     CK_KEY_TYPE keyType = CKK_GOSTR3410;
7675     CK_UTF8CHAR label[] = "A GOST R34.10-2001 cryptographic
7676         parameters object";
7677     CK_BYTE oid[] =
7678         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
7679     CK_BYTE value[] =
7680         {0x30, 0x81, 0x90, 0x02, 0x01, 0x07, 0x02, 0x20, 0x5f, 0xbff, 0xf4, 0x98,
7681         0xaa, 0x93, 0x8c, 0xe7, 0x39, 0xb8, 0xe0, 0x22, 0xfb, 0xaf, 0xef, 0x40,
7682         0x56, 0x3f, 0x6e, 0x6a, 0x34, 0x72, 0xfc, 0x2a, 0x51, 0x4c, 0x0c, 0xe9,
7683         0xda, 0xe2, 0x3b, 0x7e, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00,
7684         0x00, 0x00,
    
```

```

7685     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
7686     0x00,0x04,0x31,0x02,0x21,0x00,0x80,0x00,0x00,0x00,0x00,0x00,
7687     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x50,0xfe,
7688     0x8a,0x18,0x92,0x97,0x61,0x54,0xc5,0x9c,0xfc,0x19,0x3a,0xcc,
7689     0xf5,0xb3,0x02,0x01,0x02,0x02,0x20,0x08,0xe2,0xa8,0xa0,0xe6,
7690     0x51,0x47,0xd4,0xbd,0x63,0x16,0x03,0x0e,0x16,0xd1,0x9c,0x85,
7691     0xc9,0x7f,0x0a,0x9c,0xa2,0x67,0x12,0x2b,0x96,0xab,0xbc,0xea,
7692     0x7e,0x8f,0xc8
7693 };
7694 CK_BBOOL true = CK_TRUE;
7695 CK_ATTRIBUTE template[] = {
7696     {CKA_CLASS, &class, sizeof(class)},
7697     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7698     {CKA_TOKEN, &true, sizeof(true)},
7699     {CKA_LABEL, label, sizeof(label)-1},
7700     {CKA_OBJECT_ID, oid, sizeof(oid)},
7701     {CKA_VALUE, value, sizeof(value)}}
7702 };
7703

```

7704 2.57.5 GOST R 34.10-2001 mechanism parameters

7705 ♦ CK_GOSTR3410_KEY_WRAP_PARAMS

7706 CK_GOSTR3410_KEY_WRAP_PARAMS is a structure that provides the parameters to the
 7707 CKM_GOSTR3410_KEY_WRAP mechanism. It is defined as follows:

```

7708     typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {
7709         CK_BYTE_PTR          pWrapOID;
7710         CK ULONG             ulWrapOIDLen;
7711         CK_BYTE_PTR          pUKM;
7712         CK ULONG             ulUKMLen;
7713         CK_OBJECT_HANDLE    hKey;
7714     } CK_GOSTR3410_KEY_WRAP_PARAMS;
7715

```

7716 The fields of the structure have the following meanings:

<i>pWrapOID</i>	pointer to a data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89. If pointer takes NULL_PTR value in C_WrapKey operation then parameters are specified in object identifier of attribute CKA_GOSTR3411_PARAMS must be used. For C_UnwrapKey operation the pointer is not used and must take NULL_PTR value anytime
<i>ulWrapOIDLen</i>	length of data with DER-encoding of the object identifier indicating the data object type of GOST 28147-89
<i>pUKM</i>	pointer to a data with UKM. If pointer takes NULL_PTR value in C_WrapKey operation then random value of UKM will be used. If pointer takes non-NULL_PTR value in C_UnwrapKey operation then the pointer value will be

		compared with UKM value of wrapped key. If these two values do not match the wrapped key will be rejected
	<i>ulUKMLen</i>	length of UKM data. If <i>pUKM</i> -pointer is different from <i>NULL_PTR</i> then equal to 8
	<i>hKey</i>	key handle. Key handle of a sender for <i>C_WrapKey</i> operation. Key handle of a receiver for <i>C_UnwrapKey</i> operation. When key handle takes <i>CK_INVALID_HANDLE</i> value then an ephemeral (one time) key pair of a sender will be used

7717 ♦ **CK_GOSTR3410_DERIVE_PARAMS**

7718 **CK_GOSTR3410_DERIVE_PARAMS** is a structure that provides the parameters to the
 7719 **CKM_GOSTR3410_DERIVE** mechanism. It is defined as follows:

```
7720     typedef struct CK_GOSTR3410_DERIVE_PARAMS {
7721         CK_EC_KDF_TYPE kdf;
7722         CK_BYTE_PTR pPublicData;
7723         CK ULONG ulPublicDataLen;
7724         CK_BYTE_PTR pUKM;
7725         CK ULONG ulUKMLen;
7726     } CK_GOSTR3410_DERIVE_PARAMS;
```

7727 The fields of the structure have the following meanings:

<i>kdf</i>	additional key diversification algorithm identifier. Possible values are <i>CKD_NULL</i> and <i>CKD_CPDIVERSIFY_KDF</i> . In case of <i>CKD_NULL</i> , result of the key derivation function described in [RFC 4357], section 5.2 is used directly; In case of <i>CKD_CPDIVERSIFY_KDF</i> , the resulting key value is additionally processed with algorithm from [RFC 4357], section 6.5.
<i>pPublicData</i> ¹	pointer to data with public key of a receiver
<i>ulPublicDataLen</i>	length of data with public key of a receiver (must be 64)
<i>pUKM</i>	pointer to a UKM data
<i>ulUKMLen</i>	length of UKM data in bytes (must be 8)

7729

7730 1 Public key of a receiver is an octet string of 64 bytes long. The public key octets correspond to the concatenation of X and Y coordinates of a point. Any one of
 7731 them is 32 bytes long and represented in little endian order.

7732 **2.57.6 GOST R 34.10-2001 key pair generation**

7733 The GOST R 34.10-2001 key pair generation mechanism, denoted
 7734 **CKM_GOSTR3410_KEY_PAIR_GEN**, is a key pair generation mechanism for GOST R 34.10-2001.
 7735 This mechanism does not have a parameter.
 7736 The mechanism generates GOST R 34.10-2001 public/private key pairs with particular
 7737 GOST R 34.10-2001 domain parameters, as specified in the **CKA_GOSTR3410_PARAMS**,

7738 **CKA_GOSTR3411_PARAMS**, and **CKA_GOST28147_PARAMS** attributes of the template for the public
7739 key. Note that **CKA_GOST28147_PARAMS** attribute may not be present in the template.

7740 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
7741 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_GOSTR3410_PARAMS**,
7742 **CKA_GOSTR3411_PARAMS**, **CKA_GOST28147_PARAMS** attributes to the new private key.

7743 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7744 are not used.

7745 **2.57.7 GOST R 34.10-2001 without hashing**

7746 The GOST R 34.10-2001 without hashing mechanism, denoted **CKM_GOSTR3410**, is a mechanism for
7747 single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the
7748 part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

7749 This mechanism does not have a parameter.

7750 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes
7751 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*,
7752 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC
7753 4490] section 3.2, and [RFC 4491] section 2.2.2.

7754 The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of
7755 GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

7756 *Table 216, GOST R 34.10-2001 without hashing: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign ¹	CKK_GOSTR3410	32 bytes	64 bytes
C_Verify ¹	CKK_GOSTR3410	32 bytes	64 bytes

7757 ¹ Single-part operations only.

7758 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7759 are not used.

7760 **2.57.8 GOST R 34.10-2001 with GOST R 34.11-94**

7761 The GOST R 34.10-2001 with GOST R 34.11-94, denoted **CKM_GOSTR3410_WITH_GOSTR3411**, is a
7762 mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire
7763 GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

7764 As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating
7765 GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are
7766 specified in object identifier of attribute **CKA_GOSTR3411_PARAMS** must be used.

7767 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes
7768 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*,
7769 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC
7770 4490] section 3.2, and [RFC 4491] section 2.2.2.

7771 The input for the mechanism is signed or should be signed message of any length. Single- and multiple-
7772 part signature operations are available.

7773 *Table 217, GOST R 34.10-2001 with GOST R 34.11-94: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign	CKK_GOSTR3410	Any	64 bytes
C_Verify	CKK_GOSTR3410	Any	64 bytes

7774 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7775 are not used.

7776 2.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001

7777 GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by
7778 **CKM_GOSTR3410_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on
7779 GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation
7780 mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used.
7781 Encrypted key is a DER-encoded structure of ASN.1 *GostR3410-KeyTransport* type [RFC 4490] section
7782 4.2.

7783 It has a parameter, a **CK_GOSTR3410_KEY_WRAP_PARAMS** structure defined in section 2.57.5.

7784 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as
7785 the **CKA_VALUE** attribute of the new key.

7786 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7787 are not used.

7788 2.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys

7789 Common key derivation, denoted **CKM_GOSTR3410_DERIVE**, is a mechanism for key derivation with
7790 assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object
7791 class **CKO_DOMAIN_PARAMETERS** and key type **CKK_GOSTR3410**. An algorithm for key derivation
7792 from [RFC 4357] (section 5.2) must be used.

7793 The mechanism contributes the result as the **CKA_VALUE** attribute of the new private key. All other
7794 attributes must be specified in a template for creating private key object.

7795 2.58 ChaCha20

7796 ChaCha20 is a secret-key stream cipher described in [CHACHA].

7797 *Table 218, ChaCha20 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_CHACHA20_KEY_GEN					✓		
CKM_CHACHA20	✓					✓	

7798

7799 2.58.1 Definitions

7800 This section defines the key type “CKK_CHACHA20” for type CK_KEY_TYPE as used in the
7801 CKA_KEY_TYPE attribute of key objects.

7802 Mechanisms:

7803 CKM_CHACHA20_KEY_GEN

7804 CKM_CHACHA20

7805 2.58.2 ChaCha20 secret key objects

7806 ChaCha20 secret key objects (object class CKO_SECRET_KEY, key type CKK_CHACHA) hold
7807 ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the
7808 common attributes defined for this object class:

7809 *Table 219, ChaCha20 Secret Key Object*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key length is fixed at 256 bits. Bit length restricted to a byte array.
CKA_VALUE_LEN ^{2,3}	CK ULONG	Length in bytes of key value

7810 The following is a sample template for creating a ChaCha20 secret key object:

```
7811     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
7812     CK_KEY_TYPE keyType = CKK_CHACHA20;
7813     CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
7814     CK_BYTE value[32] = {...};
7815     CK_BBOOL true = CK_TRUE;
7816     CK_ATTRIBUTE template[] = {
7817         {CKA_CLASS, &class, sizeof(class)},
7818         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7819         {CKA_TOKEN, &true, sizeof(true)},
7820         {CKA_LABEL, label, sizeof(label)-1},
7821         {CKA_ENCRYPT, &true, sizeof(true)},
7822         {CKA_VALUE, value, sizeof(value)}}
7823     };
```

7824 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first
7825 three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.

7826 2.58.3 ChaCha20 mechanism parameters

7827 ◆ CK_CHACHA20_PARAMS; CK_CHACHA20_PARAMS_PTR

7828 CK_CHACHA20_PARAMS provides the parameters to the CKM_CHACHA20 mechanism. It is defined
7829 as follows:

```
7830     typedef struct CK_CHACHA20_PARAMS {
7831         CK_BYTE_PTR pBlockCounter;
7832         CK ULONG     blockCounterBits;
7833         CK_BYTE_PTR pNonce;
7834         CK ULONG     ulNonceBits;
7835     } CK_CHACHA20_PARAMS;
```

7836 The fields of the structure have the following meanings:

7837 pBlockCounter pointer to block counter

7838 ulblockCounterBits length of block counter in bits (can be either 32 or 64)

7839 pNonce nonce (This should be never re-used with the same key.)

7840 ulNonceBits length of nonce in bits (is 64 for original, 96 for IETF and 192 for
7841 xchacha20 variant)

7842 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)
7843 it is necessary to address these blocks in random order, thus this counter is exposed here.

7844 **2.58.4 ChaCha20 key generation**

7845 The ChaCha20 key generation mechanism, denoted **CKM_CHACHA20_KEY_GEN**, is a key generation
7846 mechanism for ChaCha20.

7847 It does not have a parameter.

7848 The mechanism generates ChaCha20 keys of 256 bits.

7849 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
7850 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
7851 supports) may be specified in the template for the key, or else are assigned default initial values.

7852 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7853 specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is
7854 fixed at 256 bits.

7855

7856 **2.58.5 ChaCha20 mechanism**

7857 ChaCha20, denoted **CKM_CHACHA20**, is a mechanism for single and multiple-part encryption and
7858 decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and
7859 handling of their nonces, affecting the safety of using random nonces and the maximum size that can be
7860 encrypted safely.

7861 Chacha20 has a parameter, **CK_CHACHA20_PARAMS**, which indicates the nonce and initial block
7862 counter value.

7863 Constraints on key types and the length of input and output data are summarized in the following table:

7864 *Table 220, ChaCha20: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	ChaCha20	Any / only up to 256 GB in case of IETF variant	Same as input length	No final part
C_Decrypt	ChaCha20	Any / only up to 256 GB in case of IETF variant	Same as input length	No final part

7865 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7866 specify the supported range of ChaCha20 key sizes, in bits.

7867 *Table 221, ChaCha20: Nonce and block counter lengths*

Variant	Nonce	Block counter	Maximum message	Nonce generation
original	64 bit	64 bit	Virtually unlimited	1 st msg: nonce ₀ =random n th msg: nonce _{n-1} ++

IETF	96 bit	32 bit	Max ~256 GB	1 st msg: nonce ₀ =random n th msg: nonce _{n-1} ++
XChaCha20	192 bit	64 bit	Virtually unlimited	Each nonce can be randomly generated.

7868 Nonces must not ever be reused with the same key. However due to the birthday paradox the first two
 7869 variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended
 7870 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.
 7871 Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with
 7872 randomly generated nonces the birthday paradox.

2.59 Salsa20

7874 Salsa20 is a secret-key stream cipher described in [SALSA].

7875 *Table 222, Salsa20 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_SALSA20_KEY_GEN					✓		
CKM_SALSA20	✓					✓	

7876

2.59.1 Definitions

7878 This section defines the key type “CKK_SALSA20” and “CKK_SALSA20” for type CK_KEY_TYPE as
 7879 used in the CKA_KEY_TYPE attribute of key objects.

7880 Mechanisms:

7881 CKM_SALSA20_KEY_GEN

7882 CKM_SALSA20

2.59.2 Salsa20 secret key objects

7884 Salsa20 secret key objects (object class CKO_SECRET_KEY, key type CKK_SALSA) hold Salsa20 keys.
 7885 The following table defines the Salsa20 secret key object attributes, in addition to the common attributes
 7886 defined for this object class:

7887 *Table 223, ChaCha20 Secret Key Object*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key length is fixed at 256 bits. Bit length restricted to a byte array.
CKA_VALUE_LEN ^{2,3}	CK ULONG	Length in bytes of key value

7888 The following is a sample template for creating a Salsa20 secret key object:

7889 CK_OBJECT_CLASS class = CKO_SECRET_KEY;

```

7890 CK_KEY_TYPE keyType = CKK_SALSA20;
7891 CK_UTF8CHAR label[] = "A Salsa20 secret key object";
7892 CK_BYTE value[32] = {...};
7893 CK_BBOOL true = CK_TRUE;
7894 CK_ATTRIBUTE template[] = {
7895     {CKA_CLASS, &class, sizeof(class)},
7896     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7897     {CKA_TOKEN, &true, sizeof(true)},
7898     {CKA_LABEL, label, sizeof(label)-1},
7899     {CKA_ENCRYPT, &true, sizeof(true)},
7900     {CKA_VALUE, value, sizeof(value)}}
7901 };
7902 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first
7903 three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.

```

7904 2.59.3 Salsa20 mechanism parameters

7905 ◆ CK_SALSA20_PARAMS; CK_SALSA_PARAMS_PTR

7906 **CK_SALSA20_PARAMS** provides the parameters to the **CKM_SALSA20** mechanism. It is defined as
7907 follows:

```

7908     typedef struct CK_SALSA20_PARAMS {
7909         CK_BYTE_PTR pBlockCounter;
7910         CK_BYTE_PTR pNonce;
7911         CK ULONG ulNonceBits;
7912     } CK_SALSA20_PARAMS;
7913

```

7914 The fields of the structure have the following meanings:

7915 *pBlockCounter* pointer to block counter (64 bits)

7916 *pNonce* nonce

7917 *ulNonceBits* size of the nonce in bits (64 for classic and 192 for XSalsa20)

7918 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)
7919 it is necessary to address these blocks in random order, thus this counter is exposed here.

7920 2.59.4 Salsa20 key generation

7921 The Salsa20 key generation mechanism, denoted **CKM_SALSA20_KEY_GEN**, is a key generation
7922 mechanism for Salsa20.

7923 It does not have a parameter.

7924 The mechanism generates Salsa20 keys of 256 bits.

7925 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
7926 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
7927 supports) may be specified in the template for the key, or else are assigned default initial values.

7928 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7929 specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed
7930 at 256 bits.

7931 2.59.5 Salsa20 mechanism

7932 Salsa20, denoted **CKM_SALSA20**, is a mechanism for single and multiple-part encryption and decryption
 7933 based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and
 7934 handling of their nonces, affecting the safety of using random nonces.

7935 Salsa20 has a parameter, **CK_SALSA20_PARAMS**, which indicates the nonce and initial block counter
 7936 value.

7937 Constraints on key types and the length of input and output data are summarized in the following table:

7938 *Table 224, Salsa20: Key and Data Length*

Function	Key type	Input length	Output length	Comments
C_Encrypt	Salsa20	Any	Same as input length	No final part
C_Decrypt	Salsa20	Any	Same as input length	No final part

7939 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 7940 specify the supported range of ChaCha20 key sizes, in bits.

7941 *Table 225, Salsa20:Nonce sizes*

Variant	Nonce	Maximum message	Nonce generation
original	64 bit	Virtually unlimited	1 st msg: nonce ₀ =random n th msg: nonce _{n-1} ++
XSalsa20	192 bit	Virtually unlimited	Each nonce can be randomly generated.

7942 Nonces must not ever be reused with the same key. However due to the birthday paradox the original
 7943 variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended
 7944 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.
 7945 Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly
 7946 generated nonces the birthday paradox.

7947 2.60 Poly1305

7948 Poly1305 is a message authentication code designed by D.J Bernstein [**POLY1305**]. Poly1305 takes a
 7949 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

7950 *Table 226, Poly1305 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_POLY1305_KEY_GEN					✓		
CKM_POLY1305		✓					

7951 2.60.1 Definitions

7952 This section defines the key type “CKK_POLY1305” for type CK_KEY_TYPE as used in the
7953 CKA_KEY_TYPE attribute of key objects.

7954 Mechanisms:

7955 CKM_POLY1305_KEY_GEN

7956 CKM_POLY1305_MAC

7957 2.60.2 Poly1305 secret key objects

7958 Poly1305 secret key objects (object class CKO_SECRET_KEY, key type CKK_POLY1305) hold
7959 Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the
7960 common attributes defined for this object class:

7961 *Table 227, Poly1305 Secret Key Object*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key length is fixed at 256 bits. Bit length restricted to a byte array.
CKA_VALUE_LEN ^{2,3}	CK ULONG	Length in bytes of key value

7962 The following is a sample template for creating a Poly1305 secret key object:

```
7963     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
7964     CK_KEY_TYPE keyType = CKK_POLY1305;
7965     CK_UTF8CHAR label[] = "A Poly1305 secret key object";
7966     CK_BYTE value[32] = {...};
7967     CK_BBOOL true = CK_TRUE;
7968     CK_ATTRIBUTE template[] = {
7969         {CKA_CLASS, &class, sizeof(class)},
7970         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7971         {CKA_TOKEN, &true, sizeof(true)},
7972         {CKA_LABEL, label, sizeof(label)-1},
7973         {CKA_SIGN, &true, sizeof(true)},
7974         {CKA_VALUE, value, sizeof(value)}}
7975     };
```

7976

7977 2.60.3 Poly1305 mechanism

7978 Poly1305, denoted **CKM_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key
7979 and arbitrary length input.

7980 It has no parameters.

7981 Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

7982 *Table 228, Poly1305: Key and Data Length*

Function	Key type	Data length	Signature Length
C_Sign	Poly1305	Any	128 bits
C_Verify	Poly1305	Any	128 bits

7983 **2.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated** 7984 **Encryption / Decryption**

7985 The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305
7986 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data
7987 (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

7988 **2.61.1 Definitions**

7989 Mechanisms:

7990 **CKM_CHACHA20_POLY1305**

7991 **CKM_SALSA20_POLY1305**

7992 **2.61.2 Usage**

7993 Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305].
7994 To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305
7995 and Salsa20/Poly1305 both use CK_SALSA20_CHACHA20_POLY1305_PARAM for Encrypt, Decrypt
7996 and CK_SALSA20_CHACHA20_POLY1305_MSG_PARAM for MessageEncrypt, and MessageDecrypt.

7997 Encrypt:

- 7998 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20
7999 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 8000 • Set the Nonce data *pNonce* in the parameter block.
- 8001 • Set the AAD data *pAAD* and size *uAADLen* in the parameter block. *pAAD* may be NULL if
8002 *uAADLen* is 0.
- 8003 • Call **C_EncryptInit()** for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
8004 mechanism with parameters and key *K*.
- 8005 • Call **C_Encrypt()**, or **C_EncryptUpdate()***¹⁰ **C_EncryptFinal()**, for the plaintext obtaining ciphertext
8006 and authentication tag output.

8007 Decrypt:

- 8008 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20
8009 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 8010 • Set the Nonce data *pNonce* in the parameter block.
- 8011 • Set the AAD data *pAAD* and size *uAADLen* in the parameter block. *pAAD* may be NULL if
8012 *uAADLen* is 0.
- 8013 • Call **C_DecryptInit()** for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
8014 mechanism with parameters and key *K*.
- 8015 • Call **C_Decrypt()**, or **C_DecryptUpdate()***¹ **C_DecryptFinal()**, for the ciphertext, including the
8016 appended tag, obtaining plaintext output. Note: since **CKM_CHACHA20_POLY1305** and
8017 **CKM_SALSA20_POLY1305** are AEAD ciphers, no data should be returned until **C_Decrypt()** or
8018 **C_DecryptFinal()**.

8019 MessageEncrypt::

10 “*” indicates 0 or more calls may be made as required

- 8020 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20
8021 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

- 8022 • Set the Nonce data *pNonce* in the parameter block.

- 8023 • Set *pTag* to hold the tag data returned from *C_EncryptMessage()* or the final
8024 *C_EncryptMessageNext()*.

- 8025 • Call *C_MessageEncryptInit()* for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
8026 mechanism with key *K*.

- 8027 • Call *C_EncryptMessage()*, or *C_EncryptMessageBegin* followed by *C_EncryptMessageNext()**¹¹.
8028 The mechanism parameter is passed to all three of these functions.

- 8029 • Call *C_MessageEncryptFinal()* to close the message decryption.

8030 MessageDecrypt:

- 8031 • Set the Nonce length *uNonceLen* in the parameter block. (this affects which variant of Chacha20
8032 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

- 8033 • Set the Nonce data *pNonce* in the parameter block.

- 8034 • Set the tag data *pTag* in the parameter block before *C_DecryptMessage* or the final
8035 *C_DecryptMessageNext()*

- 8036 • Call *C_MessageDecryptInit()* for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
8037 mechanism with key *K*.

- 8038 • Call *C_DecryptMessage()*, or *C_DecryptMessageBegin* followed by *C_DecryptMessageNext()**¹².
8039 The mechanism parameter is passed to all three of these functions.

- 8040 • Call *C_MessageDecryptFinal()* to close the message decryption

8041

8042 *uNonceLen* is the length of the nonce in bits.

8043 In Encrypt and Decrypt the tag is appended to the cipher text. In MessageEncrypt the tag is returned in
8044 the *pTag* field of **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS**. In MessageDecrypt the tag is
8045 provided by the *pTag* field of **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS**. The application
8046 must provide 16 bytes of space for the tag.

8047 The key type for *K* must be compatible with **CKM_CHACHA20** or **CKM_SALSA20** respectively and the
8048 *C_EncryptInit/C_DecryptInit* calls shall behave, with respect to *K*, as if they were called directly with
8049 **CKM_CHACHA20** or **CKM_SALSA20**, *K* and NULL parameters.

8050 Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose
8051 the block counter, as the AEAD construction is based on a message metaphor in which random access is
8052 not needed.

8053 2.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters

- 8054 ◆ **CK_SALSA20_CHACHA20_POLY1305_PARAMS;**
8055 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR**

8056 **CK_SALSA20_CHACHA20_POLY1305_PARAMS** is a structure that provides the parameters to the
8057 **CKM_CHACHA20_POLY1305** and **CKM_SALSA20_POLY1305** mechanisms. It is defined as follows:

8058

```
typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {
```

11 “*” indicates 0 or more calls may be made as required

12 “*” indicates 0 or more calls may be made as required

```

8059     CK_BYTE_PTR pNonce;
8060     CK ULONG ulNonceLen;
8061     CK_BYTE_PTR pAAD;
8062     CK ULONG ulAADLen;
8063 } CK_SALSA20_CHACHA20_POLY1305_PARAMS;
8064
8065 The fields of the structure have the following meanings:
8066     pNonce      nonce (This should be never re-used with the same key.)
8067
8068     ulNonceLen   length of nonce in bits (is 64 for original, 96 for IETF (only for
8069                           chacha20) and 192 for xchacha20/xsalsa20 variant)
8070
8071     pAAD        pointer to additional authentication data. This data is authenticated
8072                           but not encrypted.
8073
8074     ulAADLen    length of pAAD in bytes.
8075
8076 CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR is a pointer to a
8077 CK_SALSA20_CHACHA20_POLY1305_PARAMS.
8078
8079 ◆ CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
8080     CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR
8081
8082 CK_CHACHA20POLY1305_PARAMS is a structure that provides the parameters to the CKM_
8083 CHACHA20_POLY1305 mechanism. It is defined as follows:
8084
8085     typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {
8086         CK_BYTE_PTR pNonce;
8087         CK ULONG ulNonceLen;
8088         CK_BYTE_PTR pTag;
8089     } CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
8090
8091 The fields of the structure have the following meanings:
8092     pNonce      pointer to nonce
8093
8094     ulNonceLen   length of nonce in bits. The length of the influences which variant of
8095                           the ChaCha20 will be used (64 original, 96 IETF(only for
8096                           ChaCha20), 192 XChaCha20/XSalsa20)
8097
8098     pTag        location of the authentication tag which is returned on
8099                           MessageEncrypt, and provided on MessageDecrypt.
8100
8101 CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR is a pointer to a
8102 CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS.

```

2.62 HKDF Mechanisms

8093 Details for HKDF key derivation mechanisms can be found in [RFC 5869].
8094 *Table 229, HKDF Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_HKDF_DERIVE							✓
CKM_HKDF_DATA							✓
CKM_HKDF_KEY_GEN					✓		

2.62.1 Definitions

Mechanisms:

CKM_HKDF_DERIVE
 CKM_HKDF_DATA
 CKM_HKDF_KEY_GEN

Key Types:

CKK_HKDF

2.62.2 HKDF mechanism parameters

◆ CK_HKDF_PARAMS; CK_HKDF_PARAMS_PTR

CK_HKDF_PARAMS is a structure that provides the parameters to the CKM_HKDF_DERIVE and CKM_HKDF_DATA mechanisms. It is defined as follows:

```
8107     typedef struct CK_HKDF_PARAMS {
8108         CK_BOOL bExtract;
8109         CK_BOOL bExpand;
8110         CK_MECHANISM_TYPE prfHashMechanism;
8111         CK ULONG ulSaltType;
8112         CK_BYTE_PTR pSalt;
8113         CK ULONG ulSaltLen;
8114         CK_HANDLE hSaltKey;
8115         CK_BYTE_PTR pInfo;
8116         CK ULONG ulInfoLen;
8117     } CK_HKDF_PARAMS;
```

The fields of the structure have the following meanings:

8120 **bExtract** *execute the extract portion of HKDF.*

8121 **bExpand** *execute the expand portion of HKDF.*

8122 **prfHashMechanism** *base hash used for the HMAC in the underlying HKDF operation.*

8123 **ulSaltType** *specifies how the salt for the extract portion of the KDF is supplied.*

8124 *CKF_HKDF_SALT_NULL no salt is supplied.*

8125 *CKF_HKDF_SALT_DATA salt is supplied as a data in pSalt with length ulSaltLen.*

8127 *CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey.*

8128 *pSalt* *pointer to the salt.*

8129 *ulSaltLen* *length of the salt pointed to in pSalt.*

8130 *hSaltKey* *object handle to the salt key.*

8131 *pInfo* *info string for the expand stage.*

8132 *ullInfoLen* *length of the info string for the expand stage.*

8133

8134 **CK_HKDF_PARAMS_PTR** is a pointer to a **CK_HKDF_PARAMS**.

2.62.3 HKDF derive

8136 HKDF derivation implements the HKDF as specified in RFC 5869. The two booleans *bExtract* and
8137 *bExpand* control whether the extract section of the HKDF or the expand section of the HKDF is in use.
8138 It has a parameter, a **CK_HKDF_PARAMS** structure, which allows for the passing of the salt and or the
8139 expansion info. The structure contains the bools *bExtract* and *bExpand* which control whether the extract
8140 or expand portions of the HKDF is to be used. This structure is defined in Section 2.62.2.
8141 The input key must be of type **CKK_HKDF** or **CKK_GENERIC_SECRET** and the length must be the size
8142 of the underlying hash function specified in *prfHashMechanism*. The exception is a data object which has
8143 the same size as the underlying hash function, and which may be supplied as an input key. In this case
8144 *bExtract* should be true and non-null salt should be supplied.
8145 Either *bExtract* or *bExpand* must be set to true. If they are both set to true, input key is first extracted then
8146 expanded. The salt is used in the extraction stage. If *bExtract* is set to true and no salt is given, a 'zero'
8147 salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified
8148 by the RFC. If *bExpand* is set to true, **CKA_VALUE_LEN** should be set to the desired key length. If it is
8149 false **CKA_VALUE_LEN** may be set to the length of the hash, but that is not necessary as the mechanism
8150 will supply this value. The salt should be ignored if *bExtract* is false. The *pInfo* should be ignored if
8151 *bExpand* is set to false.
8152 The mechanism also contributes the **CKA_CLASS**, and **CKA_VALUE** attributes to the new key. Other
8153 attributes may be specified in the template, or else are assigned default values.
8154 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
8155 class is **CKO_SECRET_KEY**. However, since these facts are all implicit in the mechanism, there is no
8156 need to specify any of them.
8157 This mechanism has the following rules about key sensitivity and extractability:

- 8158 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
8159 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
8160 default value.
- 8161 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
8162 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
8163 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8164 **CKA_SENSITIVE** attribute.
- 8165 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
8166 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8167 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8168 value from its **CKA_EXTRACTABLE** attribute.

8169 **2.62.4 HKDF Data**

8170 HKDF Data derive mechanism, denoted **CKM_HKDF_DATA**, is identical to HKDF Derive except the
8171 output is a **CKO_DATA** object whose value is the result to the derive operation. Some tokens may restrict
8172 what data may be successfully derived based on the pInfo portion of the CK_HKDF_PARAMS. All tokens
8173 must minimally support bExtract set to true and bInfo values which start with the value "tls1.3 vi". Future
8174 additional required combinations may be specified in the profile document and applications could then
8175 query the appropriate profile before depending on the mechanism.

8176 **2.62.5 HKDF Key gen**

8177 HKDF key gen, denoted CKM_HKDF_KEYGEN generates a new random HKDF key.
8178 CKA_VALUE_LENGTH must be set in the template.

8179 **2.63 NULL Mechanism**

8180 **CKM_NULL** is a mechanism used to implement the trivial pass-through function.

8181

8182 *Table 230, CKM_NULL Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_NULL	✓	✓	✓	✓		✓	✓

¹SR = SignRecover, VR = VerifyRecover

8183

8184 **2.63.1 Definitions**

8185 Mechanisms:

8186 CKM_NULL

8187 **2.63.2 CKM_NULL mechanism parameters**

8188 CKM_NULL does not have a parameter.

8189

8190 When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

8191 When used for signing, the input data is copied to the signature. When used for signature verification, it
8192 compares the input data and the signature, and returns CKR_OK (indicating that both are identical) or
8193 CKR_SIGNATURE_INVALID.

8194 When used for digesting data, the input data is copied to the message digest.

8195 When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be
8196 wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be
8197 created.

8198 When used for deriving a key, the derived key has the same value as the base key.

8199

3 PKCS #11 Implementation Conformance

- 8200 An implementation is a conforming implementation if it meets the conditions specified in one or more
8201 server profiles specified in [PKCS11-Prof].
- 8202 If a PKCS #11 implementation claims support for a particular profile, then the implementation SHALL
8203 conform to all normative statements within the clauses specified for that profile and for any subclauses to
8204 each of those clauses.

8205 Appendix A. Acknowledgments

8206 The following individuals have participated in the creation of this specification and are gratefully
8207 acknowledged:

8208 **Participants:**

8209 List needs to be pasted in here

8210 Gil Abel, Athena Smartcard Solutions, Inc.

8211 Warren Armstrong, QuintessenceLabs

8212 Jeff Bartell, Semper Foris Solutions LLC

8213 Peter Bartok, Venafi, Inc.

8214 Anthony Berglas, Cryptsoft

8215 Joseph Brand, Semper Fortis Solutions LLC

8216 Kelley Burgin, National Security Agency

8217 Robert Burns, Thales e-Security

8218 Wan-Teh Chang, Google Inc.

8219 Hai-May Chao, Oracle

8220 Janice Cheng, Vormetric, Inc.

8221 Sangrae Cho, Electronics and Telecommunications Research Institute (ETRI)

8222 Doron Cohen, SafeNet, Inc.

8223 Fadi Cotran, Futurex

8224 Tony Cox, Cryptsoft

8225 Christopher Duane, EMC

8226 Chris Dunn, SafeNet, Inc.

8227 Valerie Fenwick, Oracle

8228 Terry Fletcher, SafeNet, Inc.

8229 Susan Gleeson, Oracle

8230 Sven Gossel, Charismathics

8231 John Green, QuintessenceLabs

8232 Robert Griffin, EMC

8233 Paul Grojean, Individual

8234 Peter Gutmann, Individual

8235 Dennis E. Hamilton, Individual

8236 Thomas Hardjono, M.I.T.

8237 Tim Hudson, Cryptsoft

8238 Gershon Janssen, Individual

8239 Seunghun Jin, Electronics and Telecommunications Research Institute (ETRI)

8240 Wang Jingman, Feitan Technologies

8241 Andrey Jivsov, Symantec Corp.

8242 Mark Joseph, P6R

8243 Stefan Kaesar, Infineon Technologies

8244 Greg Kazmierczak, Wave Systems Corp.

8245 Mark Knight, Thales e-Security
8246 Darren Krahm, Google Inc.
8247 Alex Krasnov, Infineon Technologies AG
8248 Dina Kurktchi-Nimeh, Oracle
8249 Mark Lambiase, SecureAuth Corporation
8250 Lawrence Lee, GoTrust Technology Inc.
8251 John Leiseboer, QuintessenceLabs
8252 Sean Leon, Infineon Technologies
8253 Geoffrey Li, Infineon Technologies
8254 Howie Liu, Infineon Technologies
8255 Hal Lockhart, Oracle
8256 Robert Lockhart, Thales e-Security
8257 Dale Moberg, Axway Software
8258 Darren Moffat, Oracle
8259 Valery Osheter, SafeNet, Inc.
8260 Sean Parkinson, EMC
8261 Rob Philpott, EMC
8262 Mark Powers, Oracle
8263 Ajai Puri, SafeNet, Inc.
8264 Robert Relyea, Red Hat
8265 Saikat Saha, Oracle
8266 Subhash Sankuratripati, NetApp
8267 Anthony Scarpino, Oracle
8268 Johann Schoetz, Infineon Technologies AG
8269 Rayees Shamsuddin, Wave Systems Corp.
8270 Radhika Siravara, Oracle
8271 Brian Smith, Mozilla Corporation
8272 David Smith, Venafi, Inc.
8273 Ryan Smith, Futurex
8274 Jerry Smith, US Department of Defense (DoD)
8275 Oscar So, Oracle
8276 Graham Steel, Cryptosense
8277 Michael Stevens, QuintessenceLabs
8278 Michael StJohns, Individual
8279 Jim Susoy, P6R
8280 Sander Temme, Thales e-Security
8281 Kiran Thota, VMware, Inc.
8282 Walter-John Turnes, Gemini Security Solutions, Inc.
8283 Stef Walter, Red Hat
8284 James Wang, Vormetric
8285 Jeff Webb, Dell
8286 Peng Yu, Feitian Technologies

8287 Magda Zdunkiewicz, Cryptsoft

8288 Chris Zimman, Individual

8289 **Appendix B. Manifest Constants**

8290 The following definitions can be found in the appropriate computer language files referenced on the title
8291 page of this specification. Also, refer to [PKCS11_BASE] and [PKCS11_HIST] for additional definitions.

8292

```
8293   /*
8294    * Copyright © OASIS Open 2019. All right reserved.
8295    * OASIS trademark, IPR and other policies apply.
8296    * https://www.oasis-open.org/policies-guidelines/ipr
8297    */
```

8298

8299 **B.1 Object classes**

8300 #define CKO_DATA	0x00000000
8301 #define CKO_CERTIFICATE	0x00000001
8302 #define CKO_PUBLIC_KEY	0x00000002
8303 #define CKO_PRIVATE_KEY	0x00000003
8304 #define CKO_SECRET_KEY	0x00000004
8305 #define CKO_HW_FEATURE	0x00000005
8306 #define CKO_DOMAIN_PARAMETERS	0x00000006
8307 #define CKO_MECHANISM	0x00000007
8308 #define CKO OTP KEY	0x00000008
8309 #define CKO PROFILE	0x00000009
8310 #define CKO_VENDOR_DEFINED	0x80000000

8311 **B.2 Key types**

8312 #define CKK_RSA	0x00000000
8313 #define CKK_DSA	0x00000001
8314 #define CKK_DH	0x00000002
8315 #define CKK_EC	0x00000003
8316 #define CKK_X9_42_DH	0x00000004
8317 #define CKK_KEA	0x00000005
8318 #define CKK_GENERIC_SECRET	0x00000010
8319 #define CKK_RC2	0x00000011
8320 #define CKK_RC4	0x00000012
8321 #define CKK_DES	0x00000013
8322 #define CKK_DES2	0x00000014
8323 #define CKK_DES3	0x00000015
8324 #define CKK_CAST	0x00000016
8325 #define CKK_CAST3	0x00000017
8326 #define CKK_CAST128	0x00000018
8327 #define CKK_RC5	0x00000019
8328 #define CKK_IDEA	0x0000001A
8329 #define CKK_SKIPJACK	0x0000001B

```

8330 #define CKK_BATON          0x0000001C
8331 #define CKK_JUNIPER        0x0000001D
8332 #define CKK_CDMF           0x0000001E
8333 #define CKK_AES             0x0000001F
8334 #define CKK_BLOWFISH         0x00000020
8335 #define CKK_TWOFISH          0x00000021
8336 #define CKK_SECURID          0x00000022
8337 #define CKK_HOTP             0x00000023
8338 #define CKK_ACTI              0x00000024
8339 #define CKK_CAMELLIA          0x00000025
8340 #define CKK_ARIA              0x00000026
8341 #define CKK_MD5_HMAC          0x00000027
8342 #define CKK_SHA_1_HMAC         0x00000028
8343 #define CKK_RIPEMD128_HMAC    0x00000029
8344 #define CKK_RIPEMD160_HMAC    0x0000002A
8345 #define CKK_SHA256_HMAC        0x0000002B
8346 #define CKK_SHA384_HMAC        0x0000002C
8347 #define CKK_SHA512_HMAC        0x0000002D
8348 #define CKK_SHA224_HMAC        0x0000002E
8349 #define CKK_SEED               0x0000002F
8350 #define CKK_GOSTR3410          0x00000030
8351 #define CKK_GOSTR3411          0x00000031
8352 #define CKK_GOST28147          0x00000032
8353 #define CKK_CHACHA20           0x00000033
8354 #define CKK_POLY1305           0x00000034
8355 #define CKK_AES_XTS            0x00000035
8356 #define CKK_SHA3_224_HMAC       0x00000036
8357 #define CKK_SHA3_256_HMAC       0x00000037
8358 #define CKK_SHA3_384_HMAC       0x00000038
8359 #define CKK_SHA3_512_HMAC       0x00000039
8360 #define CKK_BLAKE2B_160_HMAC     0x0000003a
8361 #define CKK_BLAKE2B_256_HMAC     0x0000003b
8362 #define CKK_BLAKE2B_384_HMAC     0x0000003c
8363 #define CKK_BLAKE2B_512_HMAC     0x0000003d
8364 #define CKK_SALSA20             0x0000003e
8365 #define CKK_X2RATCHET           0x0000003f
8366 #define CKK_EC_EDWARDS          0x00000040
8367 #define CKK_EC_MONTGOMERY         0x00000041
8368 #define CKK_HKDF                 0x00000042
8369 #define CKK_VENDOR_DEFINED        0x80000000

```

8370 B.3 Key derivation functions

```

8371 #define CKD_NULL                0x00000001
8372 #define CKD_SHA1_KDF             0x00000002
8373 #define CKD_SHA1_KDF ASN1        0x00000003
8374 #define CKD_SHA1_KDF CONCATENATE 0x00000004
8375 #define CKD_SHA224_KDF           0x00000005
8376 #define CKD_SHA256_KDF           0x00000006

```

```

8377 #define CKD_SHA384_KDF 0x00000007
8378 #define CKD_SHA512_KDF 0x00000008
8379 #define CKD_CPDIVERSIFY_KDF 0x00000009
8380 #define CKD_SHA3_224_KDF 0x0000000A
8381 #define CKD_SHA3_256_KDF 0x0000000B
8382 #define CKD_SHA3_384_KDF 0x0000000C
8383 #define CKD_SHA3_512_KDF 0x0000000D
8384 #define CKD_SHA1_KDF_SP800 0x0000000E
8385 #define CKD_SHA224_KDF_SP800 0x0000000F
8386 #define CKD_SHA256_KDF_SP800 0x00000010
8387 #define CKD_SHA384_KDF_SP800 0x00000011
8388 #define CKD_SHA512_KDF_SP800 0x00000012
8389 #define CKD_SHA3_224_KDF_SP800 0x00000013
8390 #define CKD_SHA3_256_KDF_SP800 0x00000014
8391 #define CKD_SHA3_384_KDF_SP800 0x00000015
8392 #define CKD_SHA3_512_KDF_SP800 0x00000016
8393 #define CKD_BLAKE2B_160_KDF 0x00000017
8394 #define CKD_BLAKE2B_256_KDF 0x00000018
8395 #define CKD_BLAKE2B_384_KDF 0x00000019
8396 #define CKD_BLAKE2B_512_KDF 0x0000001A

```

8397 B.4 Mechanisms

```

8398 #define CKM_RSA_PKCS_KEY_PAIR_GEN 0x00000000
8399 #define CKM_RSA_PKCS 0x00000001
8400 #define CKM_RSA_9796 0x00000002
8401 #define CKM_RSA_X_509 0x00000003
8402 #define CKM_MD2_RSA_PKCS 0x00000004
8403 #define CKM_MD5_RSA_PKCS 0x00000005
8404 #define CKM_SHA1_RSA_PKCS 0x00000006
8405 #define CKM_RIPEMD128_RSA_PKCS 0x00000007
8406 #define CKM_RIPEMD160_RSA_PKCS 0x00000008
8407 #define CKM_RSA_PKCS_OAEP 0x00000009
8408 #define CKM_RSA_X9_31_KEY_PAIR_GEN 0x0000000A
8409 #define CKM_RSA_X9_31 0x0000000B
8410 #define CKM_SHA1_RSA_X9_31 0x0000000C
8411 #define CKM_RSA_PKCS_PSS 0x0000000D
8412 #define CKM_SHA1_RSA_PKCS_PSS 0x0000000E
8413 #define CKM_DSA_KEY_PAIR_GEN 0x00000010
8414 #define CKM_DSA 0x00000011
8415 #define CKM_DSA_SHA1 0x00000012
8416 #define CKM_DSA_FIPS_G_GEN 0x00000013
8417 #define CKM_DSA_SHA224 0x00000014
8418 #define CKM_DSA_SHA256 0x00000015
8419 #define CKM_DSA_SHA384 0x00000016
8420 #define CKM_DSA_SHA512 0x00000017
8421 #define CKM_DSA_SHA3_224 0x00000018
8422 #define CKM_DSA_SHA3_256 0x00000019
8423 #define CKM_DSA_SHA3_384 0x0000001A

```

```

8424 #define CKM_DSA_SHA3_512 0x0000001B
8425 #define CKM_DH_PKCS_KEY_PAIR_GEN 0x00000020
8426 #define CKM_DH_PKCS_DERIVE 0x00000021
8427 #define CKM_X9_42_DH_KEY_PAIR_GEN 0x00000030
8428 #define CKM_X9_42_DH_DERIVE 0x00000031
8429 #define CKM_X9_42_DH_HYBRID_DERIVE 0x00000032
8430 #define CKM_X9_42_MQV_DERIVE 0x00000033
8431 #define CKM_SHA256_RSA_PKCS 0x00000040
8432 #define CKM_SHA384_RSA_PKCS 0x00000041
8433 #define CKM_SHA512_RSA_PKCS 0x00000042
8434 #define CKM_SHA256_RSA_PKCS_PSS 0x00000043
8435 #define CKM_SHA384_RSA_PKCS_PSS 0x00000044
8436 #define CKM_SHA512_RSA_PKCS_PSS 0x00000045
8437 #define CKM_SHA224_RSA_PKCS 0x00000046
8438 #define CKM_SHA224_RSA_PKCS_PSS 0x00000047
8439 #define CKM_SHA512_224 0x00000048
8440 #define CKM_SHA512_224_HMAC 0x00000049
8441 #define CKM_SHA512_224_HMAC_GENERAL 0x0000004A
8442 #define CKM_SHA512_224_KEY_DERIVATION 0x0000004B
8443 #define CKM_SHA512_256 0x0000004C
8444 #define CKM_SHA512_256_HMAC 0x0000004D
8445 #define CKM_SHA512_256_HMAC_GENERAL 0x0000004E
8446 #define CKM_SHA512_256_KEY_DERIVATION 0x0000004F
8447 #define CKM_SHA512_T 0x00000050
8448 #define CKM_SHA512_T_HMAC 0x00000051
8449 #define CKM_SHA512_T_HMAC_GENERAL 0x00000052
8450 #define CKM_SHA512_T_KEY_DERIVATION 0x00000053
8451 #define CKM_SHA3_256_RSA_PKCS 0x00000060
8452 #define CKM_SHA3_384_RSA_PKCS 0x00000061
8453 #define CKM_SHA3_512_RSA_PKCS 0x00000062
8454 #define CKM_SHA3_256_RSA_PKCS_PSS 0x00000063
8455 #define CKM_SHA3_384_RSA_PKCS_PSS 0x00000064
8456 #define CKM_SHA3_512_RSA_PKCS_PSS 0x00000065
8457 #define CKM_SHA3_224_RSA_PKCS 0x00000066
8458 #define CKM_SHA3_224_RSA_PKCS_PSS 0x00000067
8459 #define CKM_RC2_KEY_GEN 0x00000100
8460 #define CKM_RC2_ECB 0x00000101
8461 #define CKM_RC2_CBC 0x00000102
8462 #define CKM_RC2_MAC 0x00000103
8463 #define CKM_RC2_MAC_GENERAL 0x00000104
8464 #define CKM_RC2_CBC_PAD 0x00000105
8465 #define CKM_RC4_KEY_GEN 0x00000110
8466 #define CKM_RC4 0x00000111
8467 #define CKM_DES_KEY_GEN 0x00000120
8468 #define CKM_DES_ECB 0x00000121
8469 #define CKM_DES_CBC 0x00000122
8470 #define CKM_DES_MAC 0x00000123
8471 #define CKM_DES_MAC_GENERAL 0x00000124

```

```

8472 #define CKM_DES_CBC_PAD 0x00000125
8473 #define CKM_DES2_KEY_GEN 0x00000130
8474 #define CKM_DES3_KEY_GEN 0x00000131
8475 #define CKM_DES3_ECB 0x00000132
8476 #define CKM_DES3_CBC 0x00000133
8477 #define CKM_DES3_MAC 0x00000134
8478 #define CKM_DES3_MAC_GENERAL 0x00000135
8479 #define CKM_DES3_CBC_PAD 0x00000136
8480 #define CKM_DES3_CMAC_GENERAL 0x00000137
8481 #define CKM_DES3_CMAC 0x00000138
8482 #define CKM_CDMF_KEY_GEN 0x00000140
8483 #define CKM_CDMF_ECB 0x00000141
8484 #define CKM_CDMF_CBC 0x00000142
8485 #define CKM_CDMF_MAC 0x00000143
8486 #define CKM_CDMF_MAC_GENERAL 0x00000144
8487 #define CKM_CDMF_CBC_PAD 0x00000145
8488 #define CKM_DES_OFB64 0x00000150
8489 #define CKM_DES_OFB8 0x00000151
8490 #define CKM_DES_CFB64 0x00000152
8491 #define CKM_DES_CFB8 0x00000153
8492 #define CKM_MD2 0x00000200
8493 #define CKM_MD2_HMAC 0x00000201
8494 #define CKM_MD2_HMAC_GENERAL 0x00000202
8495 #define CKM_MD5 0x00000210
8496 #define CKM_MD5_HMAC 0x00000211
8497 #define CKM_MD5_HMAC_GENERAL 0x00000212
8498 #define CKM_SHA_1 0x00000220
8499 #define CKM_SHA_1_HMAC 0x00000221
8500 #define CKM_SHA_1_HMAC_GENERAL 0x00000222
8501 #define CKM_RIPEMD128 0x00000230
8502 #define CKM_RIPEMD128_HMAC 0x00000231
8503 #define CKM_RIPEMD128_HMAC_GENERAL 0x00000232
8504 #define CKM_RIPEMD160 0x00000240
8505 #define CKM_RIPEMD160_HMAC 0x00000241
8506 #define CKM_RIPEMD160_HMAC_GENERAL 0x00000242
8507 #define CKM_SHA256 0x00000250
8508 #define CKM_SHA256_HMAC 0x00000251
8509 #define CKM_SHA256_HMAC_GENERAL 0x00000252
8510 #define CKM_SHA224 0x00000255
8511 #define CKM_SHA224_HMAC 0x00000256
8512 #define CKM_SHA224_HMAC_GENERAL 0x00000257
8513 #define CKM_SHA384 0x00000260
8514 #define CKM_SHA384_HMAC 0x00000261
8515 #define CKM_SHA384_HMAC_GENERAL 0x00000262
8516 #define CKM_SHA512 0x00000270
8517 #define CKM_SHA512_HMAC 0x00000271
8518 #define CKM_SHA512_HMAC_GENERAL 0x00000272
8519 #define CKM_SECURID_KEY_GEN 0x00000280

```

8520	#define CKM_SECURID	0x00000282
8521	#define CKM_HOTP_KEY_GEN	0x00000290
8522	#define CKM_HOTP	0x00000291
8523	#define CKM_ACTI	0x000002A0
8524	#define CKM_ACTI_KEY_GEN	0x000002A1
8525	#define CKM_SHA3_256	0x000002B0
8526	#define CKM_SHA3_256_HMAC	0x000002B1
8527	#define CKM_SHA3_256_HMAC_GENERAL	0x000002B2
8528	#define CKM_SHA3_256_KEY_GEN	0x000002B3
8529	#define CKM_SHA3_224	0x000002B5
8530	#define CKM_SHA3_224_HMAC	0x000002B6
8531	#define CKM_SHA3_224_HMAC_GENERAL	0x000002B7
8532	#define CKM_SHA3_224_KEY_GEN	0x000002B8
8533	#define CKM_SHA3_384	0x000002C0
8534	#define CKM_SHA3_384_HMAC	0x000002C1
8535	#define CKM_SHA3_384_HMAC_GENERAL	0x000002C2
8536	#define CKM_SHA3_384_KEY_GEN	0x000002C3
8537	#define CKM_SHA3_512	0x000002D0
8538	#define CKM_SHA3_512_HMAC	0x000002D1
8539	#define CKM_SHA3_512_HMAC_GENERAL	0x000002D2
8540	#define CKM_SHA3_512_KEY_GEN	0x000002D3
8541	#define CKM_CAST_KEY_GEN	0x00000300
8542	#define CKM_CAST_ECB	0x00000301
8543	#define CKM_CAST_CBC	0x00000302
8544	#define CKM_CAST_MAC	0x00000303
8545	#define CKM_CAST_MAC_GENERAL	0x00000304
8546	#define CKM_CAST_CBC_PAD	0x00000305
8547	#define CKM_CAST3_KEY_GEN	0x00000310
8548	#define CKM_CAST3_ECB	0x00000311
8549	#define CKM_CAST3_CBC	0x00000312
8550	#define CKM_CAST3_MAC	0x00000313
8551	#define CKM_CAST3_MAC_GENERAL	0x00000314
8552	#define CKM_CAST3_CBC_PAD	0x00000315
8553	#define CKM_CAST128_KEY_GEN	0x00000320
8554	#define CKM_CAST128_ECB	0x00000321
8555	#define CKM_CAST128_CBC	0x00000322
8556	#define CKM_CAST128_MAC	0x00000323
8557	#define CKM_CAST128_MAC_GENERAL	0x00000324
8558	#define CKM_CAST128_CBC_PAD	0x00000325
8559	#define CKM_RC5_KEY_GEN	0x00000330
8560	#define CKM_RC5_ECB	0x00000331
8561	#define CKM_RC5_CBC	0x00000332
8562	#define CKM_RC5_MAC	0x00000333
8563	#define CKM_RC5_MAC_GENERAL	0x00000334
8564	#define CKM_RC5_CBC_PAD	0x00000335
8565	#define CKM_IDEA_KEY_GEN	0x00000340
8566	#define CKM_IDEA_ECB	0x00000341
8567	#define CKM_IDEA_CBC	0x00000342

```

8568 #define CKM_IDEA_MAC 0x00000343
8569 #define CKM_IDEA_MAC_GENERAL 0x00000344
8570 #define CKM_IDEA_CBC_PAD 0x00000345
8571 #define CKM_GENERIC_SECRET_KEY_GEN 0x00000350
8572 #define CKM_CONCATENATE_BASE_AND_KEY 0x00000360
8573 #define CKM_CONCATENATE_BASE_AND_DATA 0x00000362
8574 #define CKM_CONCATENATE_DATA_AND_BASE 0x00000363
8575 #define CKM_XOR_BASE_AND_DATA 0x00000364
8576 #define CKM_EXTRACT_KEY_FROM_KEY 0x00000365
8577 #define CKM_SSL3_PRE_MASTER_KEY_GEN 0x00000370
8578 #define CKM_SSL3_MASTER_KEY_DERIVE 0x00000371
8579 #define CKM_SSL3_KEY_AND_MAC_DERIVE 0x00000372
8580 #define CKM_SSL3_MASTER_KEY_DERIVE_DH 0x00000373
8581 #define CKM_TLS_PRE_MASTER_KEY_GEN 0x00000374
8582 #define CKM_TLS_MASTER_KEY_DERIVE 0x00000375
8583 #define CKM_TLS_KEY_AND_MAC_DERIVE 0x00000376
8584 #define CKM_TLS_MASTER_KEY_DERIVE_DH 0x00000377
8585 #define CKM_TLS_PRF 0x00000378
8586 #define CKM_SSL3_MD5_MAC 0x00000380
8587 #define CKM_SSL3_SHA1_MAC 0x00000381
8588 #define CKM_MD5_KEY_DERIVATION 0x00000390
8589 #define CKM_MD2_KEY_DERIVATION 0x00000391
8590 #define CKM_SHA1_KEY_DERIVATION 0x00000392
8591 #define CKM_SHA256_KEY_DERIVATION 0x00000393
8592 #define CKM_SHA384_KEY_DERIVATION 0x00000394
8593 #define CKM_SHA512_KEY_DERIVATION 0x00000395
8594 #define CKM_SHA224_KEY_DERIVATION 0x00000396
8595 #define CKM_SHA3_256_KEY_DERIVE 0x00000397
8596 #define CKM_SHA3_224_KEY_DERIVE 0x00000398
8597 #define CKM_SHA3_384_KEY_DERIVE 0x00000399
8598 #define CKM_SHA3_512_KEY_DERIVE 0x0000039A
8599 #define CKM_SHAKE_128_KEY_DERIVE 0x0000039B
8600 #define CKM_SHAKE_256_KEY_DERIVE 0x0000039C
8601 #define CKM_PBE_MD2_DES_CBC 0x000003A0
8602 #define CKM_PBE_MD5_DES_CBC 0x000003A1
8603 #define CKM_PBE_MD5_CAST_CBC 0x000003A2
8604 #define CKM_PBE_MD5_CAST3_CBC 0x000003A3
8605 #define CKM_PBE_MD5_CAST5_CBC 0x000003A4
8606 #define CKM_PBE_MD5_CAST128_CBC 0x000003A4
8607 #define CKM_PBE_SHA1_CAST128_CBC 0x000003A5
8608 #define CKM_PBE_SHA1_RC4_128 0x000003A6
8609 #define CKM_PBE_SHA1_RC4_40 0x000003A7
8610 #define CKM_PBE_SHA1 DES3_EDE_CBC 0x000003A8
8611 #define CKM_PBE_SHA1 DES2_EDE_CBC 0x000003A9
8612 #define CKM_PBE_SHA1_RC2_128_CBC 0x000003AA
8613 #define CKM_PBE_SHA1_RC2_40_CBC 0x000003AB
8614 #define CKM_SP800_108_COUNTER_KDF 0x000003AC
8615 #define CKM_SP800_108_FEEDBACK_KDF 0x000003AD

```

```

8616 #define CKM_SP800_108_DOUBLE_PIPELINE_KDF 0x000003AE
8617 #define CKM_PKCS5_PBKD2 0x000003B0
8618 #define CKM_PBA_SHA1_WITH_SHA1_HMAC 0x000003C0
8619 #define CKM_WTLS_PRE_MASTER_KEY_GEN 0x000003D0
8620 #define CKM_WTLS_MASTER_KEY_DERIVE 0x000003D1
8621 #define CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC 0x000003D2
8622 #define CKM_WTLS_PRF 0x000003D3
8623 #define CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE 0x000003D4
8624 #define CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE 0x000003D5
8625 #define CKM_TLS12_MAC 0x000003D8
8626 #define CKM_TLS12_KDF 0x000003D9
8627 #define CKM_TLS12_MASTER_KEY_DERIVE 0x000003E0
8628 #define CKM_TLS12_KEY_AND_MAC_DERIVE 0x000003E1
8629 #define CKM_TLS12_MASTER_KEY_DERIVE_DH 0x000003E2
8630 #define CKM_TLS12_KEY_SAFE_DERIVE 0x000003E3
8631 #define CKM_TLS_MAC 0x000003E4
8632 #define CKM_TLS_KDF 0x000003E5
8633 #define CKM_KEY_WRAP_LYNKS 0x00000400
8634 #define CKM_KEY_WRAP_SET_OAEP 0x00000401
8635 #define CKM_CMS_SIG 0x00000500
8636 #define CKM_KIP_DERIVE 0x00000510
8637 #define CKM_KIP_WRAP 0x00000511
8638 #define CKM_KIP_MAC 0x00000512
8639 #define CKM_CAMELLIA_KEY_GEN 0x00000550
8640 #define CKM_CAMELLIA_ECB 0x00000551
8641 #define CKM_CAMELLIA_CBC 0x00000552
8642 #define CKM_CAMELLIA_MAC 0x00000553
8643 #define CKM_CAMELLIA_MAC_GENERAL 0x00000554
8644 #define CKM_CAMELLIA_CBC_PAD 0x00000555
8645 #define CKM_CAMELLIA_ECB_ENCRYPT_DATA 0x00000556
8646 #define CKM_CAMELLIA_CBC_ENCRYPT_DATA 0x00000557
8647 #define CKM_CAMELLIA_CTR 0x00000558
8648 #define CKM_ARIA_KEY_GEN 0x00000560
8649 #define CKM_ARIA_ECB 0x00000561
8650 #define CKM_ARIA_CBC 0x00000562
8651 #define CKM_ARIA_MAC 0x00000563
8652 #define CKM_ARIA_MAC_GENERAL 0x00000564
8653 #define CKM_ARIA_CBC_PAD 0x00000565
8654 #define CKM_ARIA_ECB_ENCRYPT_DATA 0x00000566
8655 #define CKM_ARIA_CBC_ENCRYPT_DATA 0x00000567
8656 #define CKM_SEED_KEY_GEN 0x00000650
8657 #define CKM_SEED_ECB 0x00000651
8658 #define CKM_SEED_CBC 0x00000652
8659 #define CKM_SEED_MAC 0x00000653
8660 #define CKM_SEED_MAC_GENERAL 0x00000654
8661 #define CKM_SEED_CBC_PAD 0x00000655
8662 #define CKM_SEED_ECB_ENCRYPT_DATA 0x00000656
8663 #define CKM_SEED_CBC_ENCRYPT_DATA 0x00000657

```

8664	#define CKM_SKIPJACK_KEY_GEN	0x00001000
8665	#define CKM_SKIPJACK_ECB64	0x00001001
8666	#define CKM_SKIPJACK_CBC64	0x00001002
8667	#define CKM_SKIPJACK_OFB64	0x00001003
8668	#define CKM_SKIPJACK_CFB64	0x00001004
8669	#define CKM_SKIPJACK_CFB32	0x00001005
8670	#define CKM_SKIPJACK_CFB16	0x00001006
8671	#define CKM_SKIPJACK_CFB8	0x00001007
8672	#define CKM_SKIPJACK_WRAP	0x00001008
8673	#define CKM_SKIPJACK_PRIVATE_WRAP	0x00001009
8674	#define CKM_SKIPJACK_RELAYX	0x0000100A
8675	#define CKM_KEA_KEY_PAIR_GEN	0x00001010
8676	#define CKM_KEA_KEY_DERIVE	0x00001011
8677	#define CKM_KEA_DERIVE	0x00001012
8678	#define CKM_FORTEZZA_TIMESTAMP	0x00001020
8679	#define CKM_BATON_KEY_GEN	0x00001030
8680	#define CKM_BATON_ECB128	0x00001031
8681	#define CKM_BATON_ECB96	0x00001032
8682	#define CKM_BATON_CBC128	0x00001033
8683	#define CKM_BATON_COUNTER	0x00001034
8684	#define CKM_BATON_SHUFFLE	0x00001035
8685	#define CKM_BATON_WRAP	0x00001036
8686	#define CKM_EC_KEY_PAIR_GEN	0x00001040
8687	#define CKM_ECDSA	0x00001041
8688	#define CKM_ECDSA_SHA1	0x00001042
8689	#define CKM_ECDSA_SHA224	0x00001043
8690	#define CKM_ECDSA_SHA256	0x00001044
8691	#define CKM_ECDSA_SHA384	0x00001045
8692	#define CKM_ECDSA_SHA512	0x00001046
8693	#define CKM_ECDSA_SHA3_224	0x00001047
8694	#define CKM_ECDSA_SHA3_256	0x00001048
8695	#define CKM_ECDSA_SHA3_384	0x00001049
8696	#define CKM_ECDSA_SHA3_512	0x0000104A
8697	#define CKM_ECDH1_DERIVE	0x00001050
8698	#define CKM_ECDH1_COFACTOR_DERIVE	0x00001051
8699	#define CKM_ECMQV_DERIVE	0x00001052
8700	#define CKM_ECDH_AES_KEY_WRAP	0x00001053
8701	#define CKM_RSA_AES_KEY_WRAP	0x00001054
8702	#define CKM_EC_EDWARDS_KEY_PAIR_GEN	0x00001055
8703	#define CKM_EC_MONTGOMERY_KEY_PAIR_GEN	0x00001056
8704	#define CKM_EDDSA	0x00001057
8705	#define CKM_JUNIPER_KEY_GEN	0x00001060
8706	#define CKM_JUNIPER_ECB128	0x00001061
8707	#define CKM_JUNIPER_CBC128	0x00001062
8708	#define CKM_JUNIPER_COUNTER	0x00001063
8709	#define CKM_JUNIPER_SHUFFLE	0x00001064
8710	#define CKM_JUNIPER_WRAP	0x00001065
8711	#define CKM_FASTHASH	0x00001070

8712	#define CKM_AES_XTS	0x00001071
8713	#define CKM_AEX_XTS_KEY_GEN	0x00001072
8714	#define CKM_AES_KEY_GEN	0x00001080
8715	#define CKM_AES_ECB	0x00001081
8716	#define CKM_AES_CBC	0x00001082
8717	#define CKM_AES_MAC	0x00001083
8718	#define CKM_AES_MAC_GENERAL	0x00001084
8719	#define CKM_AES_CBC_PAD	0x00001085
8720	#define CKM_AES_CTR	0x00001086
8721	#define CKM_AES_GCM	0x00001087
8722	#define CKM_AES_CCM	0x00001088
8723	#define CKM_AES_CMAC_GENERAL	0x00001089
8724	#define CKM_AES_CMAC	0x0000108A
8725	#define CKM_AES_CTS	0x0000108B
8726	#define CKM_AES_XCBC_MAC	0x0000108C
8727	#define CKM_AES_XCBC_MAC_96	0x0000108D
8728	#define CKM_AES_GMAC	0x0000108E
8729	#define CKM_BLOWFISH_KEY_GEN	0x00001090
8730	#define CKM_BLOWFISH_CBC	0x00001091
8731	#define CKM_TWOFISH_KEY_GEN	0x00001092
8732	#define CKM_TWOFISH_CBC	0x00001093
8733	#define CKM_BLOWFISH_CBC_PAD	0x00001094
8734	#define CKM_TWOFISH_CBC_PAD	0x00001095
8735	#define CKM_DES_ECB_ENCRYPT_DATA	0x00001100
8736	#define CKM_DES_CBC_ENCRYPT_DATA	0x00001101
8737	#define CKM_DES3_ECB_ENCRYPT_DATA	0x00001102
8738	#define CKM_DES3_CBC_ENCRYPT_DATA	0x00001103
8739	#define CKM_AES_ECB_ENCRYPT_DATA	0x00001104
8740	#define CKM_AES_CBC_ENCRYPT_DATA	0x00001105
8741	#define CKM_GOSTR3410_KEY_PAIR_GEN	0x00001200
8742	#define CKM_GOSTR3410	0x00001201
8743	#define CKM_GOSTR3410_WITH_GOSTR3411	0x00001202
8744	#define CKM_GOSTR3410_KEY_WRAP	0x00001203
8745	#define CKM_GOSTR3410_DERIVE	0x00001204
8746	#define CKM_GOSTR3411	0x00001210
8747	#define CKM_GOSTR3411_HMAC	0x00001211
8748	#define CKM_GOST28147_KEY_GEN	0x00001220
8749	#define CKM_GOST28147_ECB	0x00001221
8750	#define CKM_GOST28147	0x00001222
8751	#define CKM_GOST28147_MAC	0x00001223
8752	#define CKM_GOST28147_KEY_WRAP	0x00001224
8753	#define CKM_CHACHA20_KEY_GEN	0x00001225
8754	#define CKM_CHACHA20	0x00001226
8755	#define CKM_POLY1305_KEY_GEN	0x00001227
8756	#define CKM_POLY1305	0x00001228
8757	#define CKM_DSA_PARAMETER_GEN	0x00002000
8758	#define CKM_DH_PKCS_PARAMETER_GEN	0x00002001
8759	#define CKM_X9_42_DH_PKCS_PARAMETER_GEN	0x00002002

```

8760 #define CKM_DSA_PROBABLISTIC_PARAMETER_GEN 0x00002003
8761 #define CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN 0x00002004
8762 #define CKM_AES_OFB 0x00002104
8763 #define CKM_AES_CFB64 0x00002105
8764 #define CKM_AES_CFB8 0x00002106
8765 #define CKM_AES_CFB128 0x00002107
8766 #define CKM_AES_CFB1 0x00002108
8767 #define CKM_AES_KEY_WRAP 0x00002109
8768 #define CKM_AES_KEY_WRAP_PAD 0x0000210A
8769 #define CKM_AES_KEY_WRAP_KWP 0x0000210B
8770 #define CKM_RSA_PKCS TPM_1_1 0x00004001
8771 #define CKM_RSA_PKCS_OAEP TPM_1_1 0x00004002
8772 #define CKM_SHA_1_KEY_GEN 0x00004003
8773 #define CKM_SHA224_KEY_GEN 0x00004004
8774 #define CKM_SHA256_KEY_GEN 0x00004005
8775 #define CKM_SHA384_KEY_GEN 0x00004006
8776 #define CKM_SHA512_KEY_GEN 0x00004007
8777 #define CKM_SHA512_224_KEY_GEN 0x00004008
8778 #define CKM_SHA512_256_KEY_GEN 0x00004009
8779 #define CKM_SHA512_T_KEY_GEN 0x0000400A
8780 #define CKM_NULL 0x0000400B
8781 #define CKM_BLAKE2B_160 0x0000400C
8782 #define CKM_BLAKE2B_160_HMAC 0x0000400D
8783 #define CKM_BLAKE2B_160_HMAC_GENERAL 0x0000400E
8784 #define CKM_BLAKE2B_160_KEY_DERIVE 0x0000400F
8785 #define CKM_BLAKE2B_160_KEY_GEN 0x00004010
8786 #define CKM_BLAKE2B_256 0x00004011
8787 #define CKM_BLAKE2B_256_HMAC 0x00004012
8788 #define CKM_BLAKE2B_256_HMAC_GENERAL 0x00004013
8789 #define CKM_BLAKE2B_256_KEY_DERIVE 0x00004014
8790 #define CKM_BLAKE2B_256_KEY_GEN 0x00004015
8791 #define CKM_BLAKE2B_384 0x00004016
8792 #define CKM_BLAKE2B_384_HMAC 0x00004017
8793 #define CKM_BLAKE2B_384_HMAC_GENERAL 0x00004018
8794 #define CKM_BLAKE2B_384_KEY_DERIVE 0x00004019
8795 #define CKM_BLAKE2B_384_KEY_GEN 0x0000401A
8796 #define CKM_BLAKE2B_512 0x0000401B
8797 #define CKM_BLAKE2B_512_HMAC 0x0000401C
8798 #define CKM_BLAKE2B_512_HMAC_GENERAL 0x0000401D
8799 #define CKM_BLAKE2B_512_KEY_DERIVE 0x0000401E
8800 #define CKM_BLAKE2B_512_KEY_GEN 0x0000401F
8801 #define CKM_SALSA20 0x00004020
8802 #define CKM_CHACHA20_POLY1305 0x00004021
8803 #define CKM_SALSA20_POLY1305 0x00004022
8804 #define CKM_X3DH_INITIALIZE 0x00004023
8805 #define CKM_X3DH RESPOND 0x00004024
8806 #define CKM_X2RATCHET_INITIALIZE 0x00004025
8807 #define CKM_X2RATCHET_RESPOND 0x00004026

```

```

8808 #define CKM_X2RATCHET_ENCRYPT          0x00004027
8809 #define CKM_X2RATCHET_DECRYPT          0x00004028
8810 #define CKM_XEDDSA                   0x00004029
8811 #define CKM_HKDF_DERIVE              0x0000402A
8812 #define CKM_HKDF_DATA                0x0000402B
8813 #define CKM_HKDF_KEY_GEN             0x0000402C
8814 #define CKM_VENDOR_DEFINED           0x80000000

```

8815 B.5 Attributes

```

8816 #define CKA_CLASS                      0x00000000
8817 #define CKA_TOKEN                     0x00000001
8818 #define CKA_PRIVATE                  0x00000002
8819 #define CKA_LABEL                     0x00000003
8820 #define CKA_UNIQUE_ID                0x00000004
8821 #define CKA_APPLICATION              0x00000010
8822 #define CKA_VALUE                     0x00000011
8823 #define CKA_OBJECT_ID                0x00000012
8824 #define CKA_CERTIFICATE_TYPE        0x00000080
8825 #define CKA_ISSUER                    0x00000081
8826 #define CKA_SERIAL_NUMBER            0x00000082
8827 #define CKA_AC_ISSUER                 0x00000083
8828 #define CKA_OWNER                     0x00000084
8829 #define CKA_ATTR_TYPES               0x00000085
8830 #define CKA_TRUSTED                  0x00000086
8831 #define CKA_CERTIFICATE_CATEGORY    0x00000087
8832 #define CKA_JAVA_MIDP_SECURITY_DOMAIN 0x00000088
8833 #define CKA_URL                       0x00000089
8834 #define CKA_HASH_OF_SUBJECT_PUBLIC_KEY 0x0000008A
8835 #define CKA_HASH_OF_ISSUER_PUBLIC_KEY 0x0000008B
8836 #define CKA_NAME_HASH_ALGORITHM       0x0000008C
8837 #define CKA_CHECK_VALUE               0x00000090
8838 #define CKA_KEY_TYPE                 0x00000100
8839 #define CKA_SUBJECT                  0x00000101
8840 #define CKA_ID                       0x00000102
8841 #define CKA_SENSITIVE                0x00000103
8842 #define CKA_ENCRYPT                  0x00000104
8843 #define CKA_DECRYPT                  0x00000105
8844 #define CKA_WRAP                     0x00000106
8845 #define CKA_UNWRAP                  0x00000107
8846 #define CKA_SIGN                     0x00000108
8847 #define CKA_SIGN_RECOVER             0x00000109
8848 #define CKA_VERIFY                   0x0000010A
8849 #define CKA_VERIFY_RECOVER           0x0000010B
8850 #define CKA_DERIVE                  0x0000010C
8851 #define CKA_START_DATE               0x00000110
8852 #define CKA_END_DATE                 0x00000111
8853 #define CKA_MODULUS                 0x00000120
8854 #define CKA_MODULUS_BITS             0x00000121

```

8855	#define CKA_PUBLIC_EXPONENT	0x00000122
8856	#define CKA_PRIVATE_EXPONENT	0x00000123
8857	#define CKA_PRIME_1	0x00000124
8858	#define CKA_PRIME_2	0x00000125
8859	#define CKA_EXPONENT_1	0x00000126
8860	#define CKA_EXPONENT_2	0x00000127
8861	#define CKA_COEFFICIENT	0x00000128
8862	#define CKA_PUBLIC_KEY_INFO	0x00000129
8863	#define CKA_PRIME	0x00000130
8864	#define CKA_SUBPRIME	0x00000131
8865	#define CKA_BASE	0x00000132
8866	#define CKA_PRIME_BITS	0x00000133
8867	#define CKA_SUBPRIME_BITS	0x00000134
8868	#define CKA_SUB_PRIME_BITS	CKA_SUBPRIME_BITS
8869	#define CKA_VALUE_BITS	0x00000160
8870	#define CKA_VALUE_LEN	0x00000161
8871	#define CKA_EXTRACTABLE	0x00000162
8872	#define CKA_LOCAL	0x00000163
8873	#define CKA_NEVER_EXTRACTABLE	0x00000164
8874	#define CKA_ALWAYS_SENSITIVE	0x00000165
8875	#define CKA_KEY_GEN_MECHANISM	0x00000166
8876	#define CKA_MODIFIABLE	0x00000170
8877	#define CKA_COPYABLE	0x00000171
8878	#define CKA_DESTROYABLE	0x00000172
8879	#define CKA_EC_PARAMS	0x00000180
8880	#define CKA_EC_POINT	0x00000181
8881	#define CKA_WRAP_WITH_TRUSTED	0x00000210
8882	#define CKA_WRAP_TEMPLATE (CKF_ARRAY_ATTRIBUTE 0x00000211)	
8883	#define CKA_UNWRAP_TEMPLATE (CKF_ARRAY_ATTRIBUTE 0x00000212)	
8884	#define CKA_DERIVE_TEMPLATE (CKF_ARRAY_ATTRIBUTE 0x00000213)	
8885	#define CKA OTP_FORMAT	0x00000220
8886	#define CKA OTP_LENGTH	0x00000221
8887	#define CKA OTP_TIME_INTERVAL	0x00000222
8888	#define CKA OTP_USER_FRIENDLY_MODE	0x00000223
8889	#define CKA OTP_CHALLENGE_REQUIREMENT	0x00000224
8890	#define CKA OTP_TIME_REQUIREMENT	0x00000225
8891	#define CKA OTP_COUNTER_REQUIREMENT	0x00000226
8892	#define CKA OTP_PIN_REQUIREMENT	0x00000227
8893	#define CKA OTP_USER_IDENTIFIER	0x0000022A
8894	#define CKA OTP_SERVICE_IDENTIFIER	0x0000022B
8895	#define CKA OTP_SERVICE_LOGO	0x0000022C
8896	#define CKA OTP_SERVICE_LOGO_TYPE	0x0000022D
8897	#define CKA OTP_COUNTER	0x0000022E
8898	#define CKA OTP_TIME	0x0000022F
8899	#define CKA_GOSTR3410_PARAMS	0x00000250
8900	#define CKA_GOSTR3411_PARAMS	0x00000251
8901	#define CKA_GOST28147_PARAMS	0x00000252
8902	#define CKA_HW_FEATURE_TYPE	0x00000300

```

8903 #define CKA_RESET_ON_INIT          0x00000301
8904 #define CKA_HAS_RESET            0x00000302
8905 #define CKA_PIXEL_X              0x00000400
8906 #define CKA_PIXEL_Y              0x00000401
8907 #define CKA_RESOLUTION            0x00000402
8908 #define CKA_CHAR_ROWS             0x00000403
8909 #define CKA_CHAR_COLUMNS           0x00000404
8910 #define CKA_COLOR                 0x00000405
8911 #define CKA_BITS_PER_PIXEL         0x00000406
8912 #define CKA_CHAR_SETS              0x00000480
8913 #define CKA_ENCODING_METHODS        0x00000481
8914 #define CKA_MIME_TYPES              0x00000482
8915 #define CKA_MECHANISM_TYPE         0x00000500
8916 #define CKA_REQUIRED_CMS_ATTRIBUTES 0x00000501
8917 #define CKA_DEFAULT_CMS_ATTRIBUTES   0x00000502
8918 #define CKA_SUPPORTED_CMS_ATTRIBUTES 0x00000503
8919 #define CKA_ALLOWED_MECHANISMS      (CKF_ARRAY_ATTRIBUTE|0x00000600)
8920 #define CKA_VENDOR_DEFINED           0x80000000
8921

```

B.6 Attribute constants

```

8922 #define CK_OTP_FORMAT_DECIMAL       0x00000000
8923 #define CK_OTP_FORMAT_HEXADECIMAL    0x00000001
8924 #define CK_OTP_FORMAT_ALPHANUMERIC   0x00000002
8925 #define CK_OTP_FORMAT_BINARY         0x00000003
8926
8927 #define CK_OTP_PARAM_IGNORED        0x00000000
8928 #define CK_OTP_PARAM_OPTIONAL       0x00000001
8929 #define CK_OTP_PARAM_MANDATORY      0x00000002
8930
8931 #define CK_OTP_VALUE                0x00000000
8932 #define CK_OTP_PIN                  0x00000001
8933 #define CK_OTP_CHALLENGE            0x00000002
8934 #define CK_OTP_TIME                 0x00000003
8935 #define CK_OTP_COUNTER               0x00000004
8936 #define CK_OTP_FLAGS                 0x00000005
8937 #define CK_OTP_OUTPUT_LENGTH          0x00000006
8938 #define CK_OTP_FORMAT                 0x00000007
8939

```

B.7 Other constants

```

8940 #define CKF_NEXT OTP                0x00000001
8941 #define CKF_EXCLUDE_TIME            0x00000002
8942 #define CKF_EXCLUDE_COUNTER          0x00000004
8943 #define CKF_EXCLUDE_CHALLENGE        0x00000008
8944 #define CKF_EXCLUDE_PIN              0x00000010
8945 #define CKF_USER_FRIENDLY OTP          0x00000020
8946
8947

```

```
8948 #define CKF_HKDF_SALT_NULL 0x00000001  
8949 #define CKF_HKDF_SALT_DATA 0x00000002  
8950 #define CKF_HKDF_SALT_KEY 0x00000004
```

8951 B.8 Notifications

```
8952 #define CKN OTP CHANGED 0x00000001
```

8953 B.9 Return values

```
8954 #define CKR_OK 0x00000000  
8955 #define CKR_CANCEL 0x00000001  
8956 #define CKR_HOST_MEMORY 0x00000002  
8957 #define CKR_SLOT_ID_INVALID 0x00000003  
8958 #define CKR_GENERAL_ERROR 0x00000005  
8959 #define CKR_FUNCTION_FAILED 0x00000006  
8960 #define CKR_ARGUMENTS_BAD 0x00000007  
8961 #define CKR_NO_EVENT 0x00000008  
8962 #define CKR_NEED_TO_CREATE_THREADS 0x00000009  
8963 #define CKR_CANT_LOCK 0x0000000A  
8964 #define CKR_ATTRIBUTE_READ_ONLY 0x00000010  
8965 #define CKR_ATTRIBUTE_SENSITIVE 0x00000011  
8966 #define CKR_ATTRIBUTE_TYPE_INVALID 0x00000012  
8967 #define CKR_ATTRIBUTE_VALUE_INVALID 0x00000013  
8968 #define CKR_ACTION_PROHIBITED 0x0000001B  
8969 #define CKR_DATA_INVALID 0x00000020  
8970 #define CKR_DATA_LEN_RANGE 0x00000021  
8971 #define CKR_DEVICE_ERROR 0x00000030  
8972 #define CKR_DEVICE_MEMORY 0x00000031  
8973 #define CKR_DEVICE_REMOVED 0x00000032  
8974 #define CKR_ENCRYPTED_DATA_INVALID 0x00000040  
8975 #define CKR_ENCRYPTED_DATA_LEN_RANGE 0x00000041  
8976 #define CKR_AEAD_DECRYPT_FAILED 0x00000042  
8977 #define CKR_FUNCTION_CANCELED 0x00000050  
8978 #define CKR_FUNCTION_NOT_PARALLEL 0x00000051  
8979 #define CKR_FUNCTION_NOT_SUPPORTED 0x00000054  
8980 #define CKR_KEY_HANDLE_INVALID 0x00000060  
8981 #define CKR_KEY_SIZE_RANGE 0x00000062  
8982 #define CKR_KEY_TYPE_INCONSISTENT 0x00000063  
8983 #define CKR_KEY_NOT_NEEDED 0x00000064  
8984 #define CKR_KEY_CHANGED 0x00000065  
8985 #define CKR_KEY_NEEDED 0x00000066  
8986 #define CKR_KEY_INDIGESTIBLE 0x00000067  
8987 #define CKR_KEY_FUNCTION_NOT_PERMITTED 0x00000068  
8988 #define CKR_KEY_NOT_WRAPPABLE 0x00000069  
8989 #define CKR_KEY_UNEXTRACTABLE 0x0000006A  
8990 #define CKR_MECHANISM_INVALID 0x00000070  
8991 #define CKR_MECHANISM_PARAM_INVALID 0x00000071  
8992 #define CKR_OBJECT_HANDLE_INVALID 0x00000082
```

```

8993 #define CKR_OPERATION_ACTIVE 0x00000090
8994 #define CKR_OPERATION_NOT_INITIALIZED 0x00000091
8995 #define CKR_PIN_INCORRECT 0x000000A0
8996 #define CKR_PIN_INVALID 0x000000A1
8997 #define CKR_PIN_LEN_RANGE 0x000000A2
8998 #define CKR_PIN_EXPIRED 0x000000A3
8999 #define CKR_PIN_LOCKED 0x000000A4
9000 #define CKR_SESSION_CLOSED 0x000000B0
9001 #define CKR_SESSION_COUNT 0x000000B1
9002 #define CKR_SESSION_HANDLE_INVALID 0x000000B3
9003 #define CKR_SESSION_PARALLEL_NOT_SUPPORTED 0x000000B4
9004 #define CKR_SESSION_READ_ONLY 0x000000B5
9005 #define CKR_SESSION_EXISTS 0x000000B6
9006 #define CKR_SESSION_READ_ONLY_EXISTS 0x000000B7
9007 #define CKR_SESSION_READ_WRITE_SO_EXISTS 0x000000B8
9008 #define CKR_SIGNATURE_INVALID 0x000000C0
9009 #define CKR_SIGNATURE_LEN_RANGE 0x000000C1
9010 #define CKR_TEMPLATE_INCOMPLETE 0x000000D0
9011 #define CKR_TEMPLATE_INCONSISTENT 0x000000D1
9012 #define CKR_TOKEN_NOT_PRESENT 0x000000E0
9013 #define CKR_TOKEN_NOT_RECOGNIZED 0x000000E1
9014 #define CKR_TOKEN_WRITE_PROTECTED 0x000000E2
9015 #define CKR_UNWRAPPING_KEY_HANDLE_INVALID 0x000000F0
9016 #define CKR_UNWRAPPING_KEY_SIZE_RANGE 0x000000F1
9017 #define CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT 0x000000F2
9018 #define CKR_USER_ALREADY_LOGGED_IN 0x00000100
9019 #define CKR_USER_NOT_LOGGED_IN 0x00000101
9020 #define CKR_USER_PIN_NOT_INITIALIZED 0x00000102
9021 #define CKR_USER_TYPE_INVALID 0x00000103
9022 #define CKR_USER_ANOTHER_ALREADY_LOGGED_IN 0x00000104
9023 #define CKR_USER_TOO_MANY_TYPES 0x00000105
9024 #define CKR_WAPPED_KEY_INVALID 0x00000110
9025 #define CKR_WAPPED_KEY_LEN_RANGE 0x00000112
9026 #define CKR_WAPPING_KEY_HANDLE_INVALID 0x00000113
9027 #define CKR_WAPPING_KEY_SIZE_RANGE 0x00000114
9028 #define CKR_WAPPING_KEY_TYPE_INCONSISTENT 0x00000115
9029 #define CKR_RANDOM_SEED_NOT_SUPPORTED 0x00000120
9030 #define CKR_RANDOM_NO_RNG 0x00000121
9031 #define CKR_DOMAIN_PARAMS_INVALID 0x00000130
9032 #define CKR_CURVE_NOT_SUPPORTED 0x00000140
9033 #define CKR_BUFFER_TOO_SMALL 0x00000150
9034 #define CKR_SAVED_STATE_INVALID 0x00000160
9035 #define CKR_INFORMATION_SENSITIVE 0x00000170
9036 #define CKR_STATE_UNSAVEABLE 0x00000180
9037 #define CKR_CRYPTOKI_NOT_INITIALIZED 0x00000190
9038 #define CKR_CRYPTOKI_ALREADY_INITIALIZED 0x00000191
9039 #define CKR_MUTEX_BAD 0x000001A0
9040 #define CKR_MUTEX_NOT_LOCKED 0x000001A1

```

```
9041 #define CKR_NEW_PIN_MODE          0x000001B0
9042 #define CKR_NEXT OTP             0x000001B1
9043 #define CKR_EXCEEDED_MAX_ITERATIONS 0x000001B5
9044 #define CKR_FIPS_SELF_TEST_FAILED    0x000001B6
9045 #define CKR_LIBRARY_LOAD_FAILED      0x000001B7
9046 #define CKR_PIN_TOO_WEAK           0x000001B8
9047 #define CKR_PUBLIC_KEY_INVALID       0x000001B9
9048 #define CKR_FUNCTION_REJECTED        0x00000200
9049 #define CKR_VENDOR_DEFINED          0x80000000
9050
```

Appendix C. Revision History

Revision	Date	Editor	Changes Made
Wd02	18-Jun-2017	Chris Zimman	Initial version incorporating recent changes since 2.41
WD07	16-Oct-2018	Dieter Bong	See pkcs11-curr-v3.0-updates-from-wd06-to-wd07.docx
WD08	22-Oct-2018	Dieter Bong	<p>Added references [SALSA] to section 1.3</p> <p>Split <i>GOST Mechanisms vs. Functions</i> table for GOST into separate tables for the respective GOST nnn algorithms.</p> <p>Changed format of Salsa20 to Heading2, making it section 2.59</p> <p>Removed section B.1 OTP definitions (OTP definitions have become part of the standard header files and are now covered in B.6 Attribute constants)</p> <p>Copyright updated from "2013" to "2018"</p>
WD09	26-Mar-2019	Daniel Minder	<p>Added CKF_EC_CURVENAME to table 34</p> <p>Changed CK_GCM_AEAD_PARAMS to CK_GCM_MESSAGE_PARAMS</p> <p>Reworked section 2.13 (additional AES mechanisms) during F2F</p> <p>Removed Derive for CKM_AES_GMAC in Table 80 since this is not defined in section 2.13.4</p> <p>Removed solved comments of Chris typos and formatting</p> <p>Edwards curves and RFC 8410:</p> <ul style="list-style-type: none"> - Added reference to RFC 8410 at several places in 2.3.5 - 2.3.14 - Clarified that Edwards/Montgomery curves specified with curveName are incompatible with curves specified with old (since RFC 8410 is designed like this) - Added explanation for CKM_TOKEN_RESOURCE_EXCEEDED error in 2.3.14 - Changed sample template for edwards public key objects in 2.3.5 since the parameter spec was in ecPoint instead of ecParams - Added "allowed key types" table in 2.3.17 - 2.3.20 (they were lost when copying from the proposal), but corrected it for ECDH with cofactor since this is not possible according to RFC8032.

			Corrected some formatting issues raised by Darren
WD10	29 Apr 2019	Dieter Bong	<ul style="list-style-type: none"> - Updated section Related work - Added Dieter Bong as Editor - Put year 2019 in Copyright - Added section 2.62 HKDF Mechanisms; HKDF constants in Appendix B; RFC5869 in section 1.4 Non-Normative References - Section 2.40: added CKM_NULL, removed CKM_TLS10_MAC * - Section 2.52.7.1 reference to base specification corrected - Replaced reference to [PKCS11-Base] table 10 by [PKCS11-Base] table 11 throughout whole document - Removed all occurrences of CKK_ECDSA and CKA_ECDSA_PARAMS and added notices that they are deprecated - Removed #define's for CKA_SECONDARY_AUTH, CKA_AUTH_PIN_FLAGS and CKA_ALWAYS_AUTHENTICATE - Removed #define's for CAST5 mechanisms
WD10 Rev. 2	7 May 2019	Dieter Bong	<ul style="list-style-type: none"> - Moved CKM_NULL to own section 2.63 - Removed 2 remaining occurrences of CKA_ECDSA_PARAMS
WD11	May 28, 2019	Tony Cox	<ul style="list-style-type: none"> - Final cleanup of front introductory texts and links prior to CSPRD

9052