



PKCS #11 Specification Version 3.1

Committee Specification Draft 01

16 February 2022

This stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.pdf> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.docx>

Previous stage:

N/A

Latest stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.pdf> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.docx>

Technical Committee:

OASIS PKCS 11 TC

Chair:

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

Editors:

Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH
Tony Cox (tony.cox@cryptsoft.com), Cryptsoft Pty Ltd

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-spec/v3.1/csd01/include/pkcs11-v3.1/>

Related work:

This specification replaces or supersedes:

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

This specification is related to:

- *PKCS #11 Profiles Version 3.1*. Edited by Tim Hudson. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.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 stage" 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.

Key words:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [[RFC2119](#)] and [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here.

Citation format:

When referencing this document, the following citation format should be used:

[PKCS11-Spec-v3.1]

PKCS #11 Specification Version 3.1. Edited by Dieter Bong and Tony Cox. 16 February 2022. OASIS Committee Specification Draft 01. <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/csd01/pkcs11-spec-v3.1-csd01.html>. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>.

Notices:

Copyright © OASIS Open 2022. All Rights Reserved.

Distributed under the terms of the OASIS IPR Policy, [<https://www.oasis-open.org/policies-guidelines/ipr/>]. For complete copyright information please see the full Notices section in an Appendix below.

Table of Contents

1	Introduction	18
1.1	Definitions	18
1.2	Symbols and abbreviations.....	20
1.3	Normative References	22
1.4	Non-Normative References	25
2	Platform- and compiler-dependent directives for C or C++	28
2.1	Structure packing	28
2.2	Pointer-related macros	28
3	General data types	30
3.1	General information	30
3.2	Slot and token types	31
3.3	Session types	36
3.4	Object types	37
3.5	Data types for mechanisms	41
3.6	Function types	44
3.7	Locking-related types.....	49
4	Objects	52
4.1	Creating, modifying, and copying objects	53
4.1.1	Creating objects	53
4.1.2	Modifying objects.....	54
4.1.3	Copying objects	54
4.2	Common attributes	54
4.3	Hardware Feature Objects.....	55
4.3.1	Definitions	55
4.3.2	Overview.....	55
4.3.3	Clock.....	56
4.3.3.1	Definition	56
4.3.3.2	Description.....	56
4.3.4	Monotonic Counter Objects.....	56
4.3.4.1	Definition	56
4.3.4.2	Description.....	56
4.3.5	User Interface Objects.....	56
4.3.5.1	Definition	56
4.3.5.2	Description.....	57
4.4	Storage Objects	57
4.4.1	The CKA_UNIQUE_ID attribute	58
4.5	Data objects	59
4.5.1	Definitions	59
4.5.2	Overview.....	59
4.6	Certificate objects	59
4.6.1	Definitions	59
4.6.2	Overview.....	59
4.6.3	X.509 public key certificate objects	60
4.6.4	WTLS public key certificate objects.....	62

4.6.5 X.509 attribute certificate objects	63
4.7 Key objects	64
4.7.1 Definitions	64
4.7.2 Overview	64
4.8 Public key objects	65
4.9 Private key objects	67
4.10 Secret key objects	69
4.11 Domain parameter objects	71
4.11.1 Definitions	71
4.11.2 Overview	71
4.12 Mechanism objects	72
4.12.1 Definitions	72
4.12.2 Overview	72
4.13 Profile objects	72
4.13.1 Definitions	72
4.13.2 Overview	72
5 Functions	74
5.1 Function return values	77
5.1.1 Universal Cryptoki function return values	78
5.1.2 Cryptoki function return values for functions that use a session handle	78
5.1.3 Cryptoki function return values for functions that use a token	78
5.1.4 Special return value for application-supplied callbacks	79
5.1.5 Special return values for mutex-handling functions	79
5.1.6 All other Cryptoki function return values	79
5.1.7 More on relative priorities of Cryptoki errors	84
5.1.8 Error code “gotchas”	85
5.2 Conventions for functions returning output in a variable-length buffer	85
5.3 Disclaimer concerning sample code	86
5.4 General-purpose functions	86
5.4.1 C_Initialize	86
5.4.2 C_Finalize	87
5.4.3 C_GetInfo	87
5.4.4 C_GetFunctionList	88
5.4.5 C_GetInterfaceList	89
5.4.6 C_GetInterface	90
5.5 Slot and token management functions	92
5.5.1 C_GetSlotList	92
5.5.2 C_GetSlotInfo	93
5.5.3 C_GetTokenInfo	94
5.5.4 C_WaitForSlotEvent	95
5.5.5 C_GetMechanismList	96
5.5.6 C_GetMechanismInfo	97
5.5.7 C_InitToken	97
5.5.8 C_InitPIN	99
5.5.9 C_SetPIN	99
5.6 Session management functions	100

5.6.1 C_OpenSession	101
5.6.2 C_CloseSession	101
5.6.3 C_CloseAllSessions	102
5.6.4 C_GetSessionInfo	103
5.6.5 C_SessionCancel	103
5.6.6 C_GetOperationState	105
5.6.7 C_SetOperationState	106
5.6.8 C_Login	108
5.6.9 C_LoginUser	109
5.6.10 C_Logout	110
5.7 Object management functions	111
5.7.1 C_CreateObject	111
5.7.2 C_CopyObject	113
5.7.3 C_DestroyObject	114
5.7.4 C_GetObjectSize	115
5.7.5 C_GetAttributeValue	116
5.7.6 C_SetAttributeValue	118
5.7.7 C_FindObjectsInit	119
5.7.8 C_FindObjects	119
5.7.9 C_FindObjectsFinal	120
5.8 Encryption functions	120
5.8.1 C_EncryptInit	120
5.8.2 C_Encrypt	121
5.8.3 C_EncryptUpdate	122
5.8.4 C_EncryptFinal	122
5.9 Message-based encryption functions	124
5.9.1 C_MessageEncryptInit	124
5.9.2 C_EncryptMessage	125
5.9.3 C_EncryptMessageBegin	125
5.9.4 C_EncryptMessageNext	126
5.9.5 C_MessageEncryptFinal	127
5.10 Decryption functions	129
5.10.1 C_DecryptInit	129
5.10.2 C_Decrypt	129
5.10.3 C_DecryptUpdate	130
5.10.4 C_DecryptFinal	131
5.11 Message-based decryption functions	132
5.11.1 C_MessageDecryptInit	132
5.11.2 C_DecryptMessage	133
5.11.3 C_DecryptMessageBegin	134
5.11.4 C_DecryptMessageNext	134
5.11.5 C_MessageDecryptFinal	135
5.12 Message digesting functions	135
5.12.1 C_DigestInit	136
5.12.2 C_Digest	136
5.12.3 C_DigestUpdate	137

5.12.4 C_DigestKey	137
5.12.5 C_DigestFinal	137
5.13 Signing and MACing functions	139
5.13.1 C_SignInit	139
5.13.2 C_Sign	139
5.13.3 C_SignUpdate	140
5.13.4 C_SignFinal	140
5.13.5 C_SignRecoverInit	141
5.13.6 C_SignRecover	142
5.14 Message-based signing and MACing functions	143
5.14.1 C_MessageSignInit	143
5.14.2 C_SignMessage	143
5.14.3 C_SignMessageBegin	144
5.14.4 C_SignMessageNext	145
5.14.5 C_MessageSignFinal	145
5.15 Functions for verifying signatures and MACs	146
5.15.1 C_VerifyInit	146
5.15.2 C_Verify	146
5.15.3 C_VerifyUpdate	147
5.15.4 C_VerifyFinal	147
5.15.5 C_VerifyRecoverInit	148
5.15.6 C_VerifyRecover	149
5.16 Message-based functions for verifying signatures and MACs	150
5.16.1 C_MessageVerifyInit	150
5.16.2 C_VerifyMessage	150
5.16.3 C_VerifyMessageBegin	151
5.16.4 C_VerifyMessageNext	152
5.16.5 C_MessageVerifyFinal	152
5.17 Dual-function cryptographic functions	153
5.17.1 C_DigestEncryptUpdate	153
5.17.2 C_DecryptDigestUpdate	155
5.17.3 C_SignEncryptUpdate	158
5.17.4 C_DecryptVerifyUpdate	161
5.18 Key management functions	163
5.18.1 C_GenerateKey	163
5.18.2 C_GenerateKeyPair	165
5.18.3 C_WrapKey	166
5.18.4 C_UnwrapKey	168
5.18.5 C_DeriveKey	169
5.19 Random number generation functions	171
5.19.1 C_SeedRandom	171
5.19.2 C_GenerateRandom	172
5.20 Parallel function management functions	173
5.20.1 C_GetFunctionStatus	173
5.20.2 C_CancelFunction	173
5.21 Callback functions	173

5.21.1 Surrender callbacks	173
5.21.2 Vendor-defined callbacks	174
6 Mechanisms	175
6.1 RSA	175
6.1.1 Definitions	175
6.1.2 RSA public key objects	176
6.1.3 RSA private key objects	177
6.1.4 PKCS #1 RSA key pair generation	179
6.1.5 X9.31 RSA key pair generation	179
6.1.6 PKCS #1 v1.5 RSA	179
6.1.7 PKCS #1 RSA OAEP mechanism parameters	180
6.1.8 PKCS #1 RSA OAEP	182
6.1.9 PKCS #1 RSA PSS mechanism parameters	182
6.1.10 PKCS #1 RSA PSS	183
6.1.11 ISO/IEC 9796 RSA	183
6.1.12 X.509 (raw) RSA	184
6.1.13 ANSI X9.31 RSA	185
6.1.14 PKCS #1 v1.5 RSA signature with MD2, MD5, SHA-1, SHA-256, SHA-384, SHA-512, RIPE- MD 128 or RIPE-MD 160	186
6.1.15 PKCS #1 v1.5 RSA signature with SHA-224	186
6.1.16 PKCS #1 RSA PSS signature with SHA-224	186
6.1.17 PKCS #1 RSA PSS signature with SHA-1, SHA-256, SHA-384 or SHA-512	186
6.1.18 PKCS #1 v1.5 RSA signature with SHA3	187
6.1.19 PKCS #1 RSA PSS signature with SHA3	187
6.1.20 ANSI X9.31 RSA signature with SHA-1	187
6.1.21 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA	188
6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP	188
6.1.23 RSA AES KEY WRAP	189
6.1.24 RSA AES KEY WRAP mechanism parameters	190
6.1.25 FIPS 186-4	190
6.2 DSA	190
6.2.1 Definitions	191
6.2.2 DSA public key objects	192
6.2.3 DSA Key Restrictions	193
6.2.4 DSA private key objects	193
6.2.5 DSA domain parameter objects	194
6.2.6 DSA key pair generation	195
6.2.7 DSA domain parameter generation	195
6.2.8 DSA probabilistic domain parameter generation	195
6.2.9 DSA Shawe-Taylor domain parameter generation	196
6.2.10 DSA base domain parameter generation	196
6.2.11 DSA without hashing	196
6.2.12 DSA with SHA-1	197
6.2.13 FIPS 186-4	197
6.2.14 DSA with SHA-224	197
6.2.15 DSA with SHA-256	198

6.2.16 DSA with SHA-384	198
6.2.17 DSA with SHA-512	199
6.2.18 DSA with SHA3-224	199
6.2.19 DSA with SHA3-256	200
6.2.20 DSA with SHA3-384	200
6.2.21 DSA with SHA3-512	200
6.3 Elliptic Curve	201
6.3.1 EC Signatures	203
6.3.2 Definitions	203
6.3.3 Short Weierstrass Elliptic Curve public key objects	204
6.3.4 Short Weierstrass Elliptic Curve private key objects	205
6.3.5 Edwards Elliptic Curve public key objects	206
6.3.6 Edwards Elliptic Curve private key objects	207
6.3.7 Montgomery Elliptic Curve public key objects	208
6.3.8 Montgomery Elliptic Curve private key objects	209
6.3.9 Elliptic Curve key pair generation	210
6.3.10 Edwards Elliptic Curve key pair generation	211
6.3.11 Montgomery Elliptic Curve key pair generation	211
6.3.12 ECDSA without hashing	212
6.3.13 ECDSA with hashing	212
6.3.14 EdDSA	213
6.3.15 XEdDSA	213
6.3.16 EC mechanism parameters	214
6.3.17 Elliptic Curve Diffie-Hellman key derivation	219
6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation	219
6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation	220
6.3.20 ECDH AES KEY WRAP	221
6.3.21 ECDH AES KEY WRAP mechanism parameters	222
6.3.22 FIPS 186-4	223
6.4 Diffie-Hellman	223
6.4.1 Definitions	223
6.4.2 Diffie-Hellman public key objects	224
6.4.3 X9.42 Diffie-Hellman public key objects	224
6.4.4 Diffie-Hellman private key objects	225
6.4.5 X9.42 Diffie-Hellman private key objects	226
6.4.6 Diffie-Hellman domain parameter objects	227
6.4.7 X9.42 Diffie-Hellman domain parameters objects	227
6.4.8 PKCS #3 Diffie-Hellman key pair generation	228
6.4.9 PKCS #3 Diffie-Hellman domain parameter generation	229
6.4.10 PKCS #3 Diffie-Hellman key derivation	229
6.4.11 X9.42 Diffie-Hellman mechanism parameters	230
6.4.12 X9.42 Diffie-Hellman key pair generation	232
6.4.13 X9.42 Diffie-Hellman domain parameter generation	233
6.4.14 X9.42 Diffie-Hellman key derivation	233
6.4.15 X9.42 Diffie-Hellman hybrid key derivation	234
6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation	234

6.5 Extended Triple Diffie-Hellman (x3dh).....	235
6.5.1 Definitions.....	235
6.5.2 Extended Triple Diffie-Hellman key objects	235
6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange.....	235
6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange	236
6.5.5 Extended Triple Diffie-Hellman parameters	237
6.6 Double Ratchet	238
6.6.1 Definitions.....	238
6.6.2 Double Ratchet secret key objects.....	238
6.6.3 Double Ratchet key derivation	239
6.6.4 Double Ratchet Encryption mechanism	240
6.6.5 Double Ratchet parameters	241
6.7 Wrapping/unwrapping private keys	241
6.8 Generic secret key	243
6.8.1 Definitions.....	244
6.8.2 Generic secret key objects	244
6.8.3 Generic secret key generation	245
6.9 HMAC mechanisms	245
6.9.1 General block cipher mechanism parameters.....	245
6.10 AES.....	245
6.10.1 Definitions.....	246
6.10.2 AES secret key objects	246
6.10.3 AES key generation.....	247
6.10.4 AES-ECB.....	247
6.10.5 AES-CBC.....	248
6.10.6 AES-CBC with PKCS padding	248
6.10.7 AES-OFB.....	249
6.10.8 AES-CFB	249
6.10.9 General-length AES-MAC	250
6.10.10 AES-MAC	250
6.10.11 AES-XCBC-MAC	250
6.10.12 AES-XCBC-MAC-96.....	251
6.11 AES with Counter	251
6.11.1 Definitions.....	251
6.11.2 AES with Counter mechanism parameters	251
6.11.3 AES with Counter Encryption / Decryption.....	252
6.12 AES CBC with Cipher Text Stealing CTS.....	252
6.12.1 Definitions.....	252
6.12.2 AES CTS mechanism parameters	253
6.13 Additional AES Mechanisms	253
6.13.1 Definitions.....	253
6.13.2 AES-GCM Authenticated Encryption / Decryption	253
6.13.3 AES-CCM authenticated Encryption / Decryption.....	255
6.13.4 AES-GMAC	257
6.13.5 AES GCM and CCM Mechanism parameters	257
6.14 AES CMAC	260

6.14.1 Definitions	260
6.14.2 Mechanism parameters	260
6.14.3 General-length AES-CMAC	261
6.14.4 AES-CMAC	261
6.15 AES XTS	261
6.15.1 Definitions	262
6.15.2 AES-XTS secret key objects	262
6.15.3 AES-XTS key generation	262
6.15.4 AES-XTS	262
6.16 AES Key Wrap	262
6.16.1 Definitions	263
6.16.2 AES Key Wrap Mechanism parameters	263
6.16.3 AES Key Wrap	263
6.17 Key derivation by data encryption – DES & AES	264
6.17.1 Definitions	264
6.17.2 Mechanism Parameters	265
6.17.3 Mechanism Description	265
6.18 Double and Triple-length DES	265
6.18.1 Definitions	266
6.18.2 DES2 secret key objects	266
6.18.3 DES3 secret key objects	267
6.18.4 Double-length DES key generation	267
6.18.5 Triple-length DES Order of Operations	268
6.18.6 Triple-length DES in CBC Mode	268
6.18.7 DES and Triple length DES in OFB Mode	268
6.18.8 DES and Triple length DES in CFB Mode	269
6.19 Double and Triple-length DES CMAC	269
6.19.1 Definitions	269
6.19.2 Mechanism parameters	269
6.19.3 General-length DES3-MAC	270
6.19.4 DES3-CMAC	270
6.20 SHA-1	270
6.20.1 Definitions	271
6.20.2 SHA-1 digest	271
6.20.3 General-length SHA-1-HMAC	271
6.20.4 SHA-1-HMAC	272
6.20.5 SHA-1 key derivation	272
6.20.6 SHA-1 HMAC key generation	272
6.21 SHA-224	273
6.21.1 Definitions	273
6.21.2 SHA-224 digest	273
6.21.3 General-length SHA-224-HMAC	273
6.21.4 SHA-224-HMAC	274
6.21.5 SHA-224 key derivation	274
6.21.6 SHA-224 HMAC key generation	274
6.22 SHA-256	274

6.22.1 Definitions	275
6.22.2 SHA-256 digest	275
6.22.3 General-length SHA-256-HMAC	275
6.22.4 SHA-256-HMAC	276
6.22.5 SHA-256 key derivation.....	276
6.22.6 SHA-256 HMAC key generation.....	276
6.23 SHA-384	276
6.23.1 Definitions	276
6.23.2 SHA-384 digest	277
6.23.3 General-length SHA-384-HMAC	277
6.23.4 SHA-384-HMAC	277
6.23.5 SHA-384 key derivation.....	277
6.23.6 SHA-384 HMAC key generation.....	278
6.24 SHA-512	278
6.24.1 Definitions	278
6.24.2 SHA-512 digest	278
6.24.3 General-length SHA-512-HMAC	279
6.24.4 SHA-512-HMAC	279
6.24.5 SHA-512 key derivation.....	279
6.24.6 SHA-512 HMAC key generation.....	279
6.25 SHA-512/224	280
6.25.1 Definitions	280
6.25.2 SHA-512/224 digest	280
6.25.3 General-length SHA-512/224-HMAC	280
6.25.4 SHA-512/224-HMAC	281
6.25.5 SHA-512/224 key derivation.....	281
6.25.6 SHA-512/224 HMAC key generation	281
6.26 SHA-512/256	281
6.26.1 Definitions	282
6.26.2 SHA-512/256 digest	282
6.26.3 General-length SHA-512/256-HMAC	282
6.26.4 SHA-512/256-HMAC	283
6.26.5 SHA-512/256 key derivation.....	283
6.26.6 SHA-512/256 HMAC key generation	283
6.27 SHA-512/t	283
6.27.1 Definitions	284
6.27.2 SHA-512/t digest	284
6.27.3 General-length SHA-512/t-HMAC	284
6.27.4 SHA-512/t-HMAC	284
6.27.5 SHA-512/t key derivation.....	285
6.27.6 SHA-512/t HMAC key generation	285
6.28 SHA3-224	285
6.28.1 Definitions	285
6.28.2 SHA3-224 digest	286
6.28.3 General-length SHA3-224-HMAC	286
6.28.4 SHA3-224-HMAC	286

6.28.5 SHA3-224 key derivation.....	286
6.28.6 SHA3-224 HMAC key generation	286
6.29 SHA3-256	287
6.29.1 Definitions.....	287
6.29.2 SHA3-256 digest	287
6.29.3 General-length SHA3-256-HMAC	287
6.29.4 SHA3-256-HMAC	288
6.29.5 SHA3-256 key derivation.....	288
6.29.6 SHA3-256 HMAC key generation	288
6.30 SHA3-384	288
6.30.1 Definitions.....	289
6.30.2 SHA3-384 digest	289
6.30.3 General-length SHA3-384-HMAC	289
6.30.4 SHA3-384-HMAC	290
6.30.5 SHA3-384 key derivation.....	290
6.30.6 SHA3-384 HMAC key generation	290
6.31 SHA3-512	290
6.31.1 Definitions.....	290
6.31.2 SHA3-512 digest	291
6.31.3 General-length SHA3-512-HMAC	291
6.31.4 SHA3-512-HMAC	291
6.31.5 SHA3-512 key derivation.....	291
6.31.6 SHA3-512 HMAC key generation	291
6.32 SHAKE.....	292
6.32.1 Definitions.....	292
6.32.2 SHAKE Key Derivation.....	292
6.33 BLAKE2B-160.....	293
6.33.1 Definitions.....	293
6.33.2 BLAKE2B-160 digest.....	293
6.33.3 General-length BLAKE2B-160-HMAC	293
6.33.4 BLAKE2B-160-HMAC	294
6.33.5 BLAKE2B-160 key derivation.....	294
6.33.6 BLAKE2B-160 HMAC key generation.....	294
6.34 BLAKE2B-256.....	294
6.34.1 Definitions.....	295
6.34.2 BLAKE2B-256 digest.....	295
6.34.3 General-length BLAKE2B-256-HMAC	295
6.34.4 BLAKE2B-256-HMAC	295
6.34.5 BLAKE2B-256 key derivation.....	295
6.34.6 BLAKE2B-256 HMAC key generation.....	296
6.35 BLAKE2B-384.....	296
6.35.1 Definitions.....	296
6.35.2 BLAKE2B-384 digest.....	296
6.35.3 General-length BLAKE2B-384-HMAC	297
6.35.4 BLAKE2B-384-HMAC	297
6.35.5 BLAKE2B-384 key derivation.....	297

6.35.6 BLAKE2B-384 HMAC key generation.....	297
6.36 BLAKE2B-512.....	297
6.36.1 Definitions.....	298
6.36.2 BLAKE2B-512 digest.....	298
6.36.3 General-length BLAKE2B-512-HMAC	298
6.36.4 BLAKE2B-512-HMAC	299
6.36.5 BLAKE2B-512 key derivation.....	299
6.36.6 BLAKE2B-512 HMAC key generation.....	299
6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE).....	299
6.37.1 Definitions.....	300
6.37.2 Password-based encryption/authentication mechanism parameters.....	300
6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters	301
6.37.4 PKCS #5 PBKDF2 key generation	303
6.38 PKCS #12 password-based encryption/authentication mechanisms	303
6.38.1 SHA-1-PBE for 3-key triple-DES-CBC	304
6.38.2 SHA-1-PBE for 2-key triple-DES-CBC	304
6.38.3 SHA-1-PBE for SHA-1-HMAC.....	304
6.39 SSL	304
6.39.1 Definitions.....	305
6.39.2 SSL mechanism parameters	305
6.39.3 Pre-master key generation	307
6.39.4 Master key derivation	307
6.39.5 Master key derivation for Diffie-Hellman	308
6.39.6 Key and MAC derivation.....	309
6.39.7 MD5 MACing in SSL 3.0	310
6.39.8 SHA-1 MACing in SSL 3.0	310
6.40 TLS 1.2 Mechanisms	310
6.40.1 Definitions.....	311
6.40.2 TLS 1.2 mechanism parameters	311
6.40.3 TLS MAC	314
6.40.4 Master key derivation	314
6.40.5 Master key derivation for Diffie-Hellman	315
6.40.6 Key and MAC derivation.....	316
6.40.7 CKM_TLS12_KEY_SAFE_DERIVE.....	316
6.40.8 Generic Key Derivation using the TLS PRF.....	317
6.40.9 Generic Key Derivation using the TLS12 PRF.....	317
6.41 WTLS.....	318
6.41.1 Definitions.....	318
6.41.2 WTLS mechanism parameters.....	319
6.41.3 Pre master secret key generation for RSA key exchange suite.....	321
6.41.4 Master secret key derivation	322
6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography	322
6.41.6 WTLS PRF (pseudorandom function)	323
6.41.7 Server Key and MAC derivation.....	323
6.41.8 Client key and MAC derivation	324
6.42 SP 800-108 Key Derivation	325

6.42.1 Definitions	325
6.42.2 Mechanism Parameters	326
6.42.3 Counter Mode KDF	331
6.42.4 Feedback Mode KDF	331
6.42.5 Double Pipeline Mode KDF	332
6.42.6 Deriving Additional Keys	333
6.42.7 Key Derivation Attribute Rules	334
6.42.8 Constructing PRF Input Data	334
6.42.8.1 Sample Counter Mode KDF	334
6.42.8.2 Sample SCP03 Counter Mode KDF	335
6.42.8.3 Sample Feedback Mode KDF	336
6.42.8.4 Sample Double-Pipeline Mode KDF	338
6.43 Miscellaneous simple key derivation mechanisms	339
6.43.1 Definitions	339
6.43.2 Parameters for miscellaneous simple key derivation mechanisms	339
6.43.3 Concatenation of a base key and another key	340
6.43.4 Concatenation of a base key and data	340
6.43.5 Concatenation of data and a base key	341
6.43.6 XORing of a key and data	342
6.43.7 Extraction of one key from another key	343
6.44 CMS	343
6.44.1 Definitions	344
6.44.2 CMS Signature Mechanism Objects	344
6.44.3 CMS mechanism parameters	344
6.44.4 CMS signatures	345
6.45 Blowfish	346
6.45.1 Definitions	347
6.45.2 BLOWFISH secret key objects	347
6.45.3 Blowfish key generation	348
6.45.4 Blowfish-CBC	348
6.45.5 Blowfish-CBC with PKCS padding	348
6.46 Twofish	349
6.46.1 Definitions	349
6.46.2 Twofish secret key objects	349
6.46.3 Twofish key generation	350
6.46.4 Twofish -CBC	350
6.46.5 Twofish-CBC with PKCS padding	350
6.47 CAMELLIA	350
6.47.1 Definitions	351
6.47.2 Camellia secret key objects	351
6.47.3 Camellia key generation	352
6.47.4 Camellia-ECB	352
6.47.5 Camellia-CBC	353
6.47.6 Camellia-CBC with PKCS padding	353
6.47.7 CAMELLIA with Counter mechanism parameters	354
6.47.8 General-length Camellia-MAC	355
6.47.9 Camellia-MAC	355

6.48 Key derivation by data encryption - Camellia	356
6.48.1 Definitions	356
6.48.2 Mechanism Parameters	356
6.49 ARIA.....	356
6.49.1 Definitions	357
6.49.2 Aria secret key objects	357
6.49.3 ARIA key generation	358
6.49.4 ARIA-ECB.....	358
6.49.5 ARIA-CBC	358
6.49.6 ARIA-CBC with PKCS padding	359
6.49.7 General-length ARIA-MAC	360
6.49.8 ARIA-MAC	360
6.50 Key derivation by data encryption - ARIA.....	360
6.50.1 Definitions	361
6.50.2 Mechanism Parameters	361
6.51 SEED	361
6.51.1 Definitions	362
6.51.2 SEED secret key objects.....	362
6.51.3 SEED key generation	363
6.51.4 SEED-ECB	363
6.51.5 SEED-CBC	363
6.51.6 SEED-CBC with PKCS padding.....	363
6.51.7 General-length SEED-MAC.....	364
6.51.8 SEED-MAC.....	364
6.52 Key derivation by data encryption - SEED	364
6.52.1 Definitions	364
6.52.2 Mechanism Parameters	364
6.53 OTP.....	365
6.53.1 Usage overview	365
6.53.2 Case 1: Generation of OTP values	365
6.53.3 Case 2: Verification of provided OTP values	366
6.53.4 Case 3: Generation of OTP keys	366
6.53.5 OTP objects.....	367
6.53.5.1 Key objects	367
6.53.6 OTP-related notifications.....	370
6.53.7 OTP mechanisms	370
6.53.7.1 OTP mechanism parameters	370
6.53.8 RSA SecurID	374
6.53.8.1 RSA SecurID secret key objects	374
6.53.8.2 RSA SecurID key generation	375
6.53.8.3 SecurID OTP generation and validation.....	375
6.53.8.4 Return values.....	376
6.53.9 OATH HOTP.....	376
6.53.9.1 OATH HOTP secret key objects	376
6.53.9.2 HOTP key generation	377
6.53.9.3 HOTP OTP generation and validation.....	377
6.53.10 ActivIdentity ACTI.....	377

6.53.10.1 ACTI secret key objects	377
6.53.10.2 ACTI key generation	378
6.53.10.3 ACTI OTP generation and validation	378
6.54 CT-KIP	379
6.54.1 Principles of Operation	379
6.54.2 Mechanisms	379
6.54.3 Definitions	380
6.54.4 CT-KIP Mechanism parameters	380
6.54.5 CT-KIP key derivation	380
6.54.6 CT-KIP key wrap and key unwrap	381
6.54.7 CT-KIP signature generation	381
6.55 GOST 28147-89	381
6.55.1 Definitions	381
6.55.2 GOST 28147-89 secret key objects	382
6.55.3 GOST 28147-89 domain parameter objects	382
6.55.4 GOST 28147-89 key generation	383
6.55.5 GOST 28147-89-ECB	384
6.55.6 GOST 28147-89 encryption mode except ECB	384
6.55.7 GOST 28147-89-MAC	385
6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89	385
6.56 GOST R 34.11-94	386
6.56.1 Definitions	386
6.56.2 GOST R 34.11-94 domain parameter objects	386
6.56.3 GOST R 34.11-94 digest	387
6.56.4 GOST R 34.11-94 HMAC	388
6.57 GOST R 34.10-2001	388
6.57.1 Definitions	388
6.57.2 GOST R 34.10-2001 public key objects	389
6.57.3 GOST R 34.10-2001 private key objects	390
6.57.4 GOST R 34.10-2001 domain parameter objects	392
6.57.5 GOST R 34.10-2001 mechanism parameters	393
6.57.6 GOST R 34.10-2001 key pair generation	394
6.57.7 GOST R 34.10-2001 without hashing	394
6.57.8 GOST R 34.10-2001 with GOST R 34.11-94	395
6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001	395
6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys	395
6.58 ChaCha20	396
6.58.1 Definitions	396
6.58.2 ChaCha20 secret key objects	396
6.58.3 ChaCha20 mechanism parameters	397
6.58.4 ChaCha20 key generation	397
6.58.5 ChaCha20 mechanism	397
6.59 Salsa20	398
6.59.1 Definitions	399
6.59.2 Salsa20 secret key objects	399
6.59.3 Salsa20 mechanism parameters	400

6.59.4 Salsa20 key generation	400
6.59.5 Salsa20 mechanism	400
6.60 Poly1305	401
6.60.1 Definitions	401
6.60.2 Poly1305 secret key objects	401
6.60.3 Poly1305 mechanism	402
6.61 ChaCha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption	402
6.61.1 Definitions	403
6.61.2 Usage	403
6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters	404
6.62 HKDF Mechanisms	405
6.62.1 Definitions	406
6.62.2 HKDF mechanism parameters	406
6.62.3 HKDF derive	407
6.62.4 HKDF Data	407
6.62.5 HKDF Key gen	407
6.63 NULL Mechanism	407
6.63.1 Definitions	408
6.63.2 CKM_NULL mechanism parameters	408
6.64 IKE Mechanisms	408
6.64.1 Definitions	408
6.64.2 IKE mechanism parameters	409
6.64.3 IKE PRF DERIVE	411
6.64.4 IKEv1 PRF DERIVE	412
6.64.5 IKEv2 PRF PLUS DERIVE	412
6.64.6 IKEv1 Extended Derive	413
6.65 HSS	413
6.65.1 Definitions	414
6.65.2 HSS public key objects	414
6.65.3 HSS private key objects	415
6.65.4 HSS key pair generation	416
6.65.5 HSS without hashing	416
7 PKCS #11 Implementation Conformance	418
7.1 PKCS#11 Consumer Implementation Conformance	418
7.2 PKCS#11 Provider Implementation Conformance	418
Appendix A. Acknowledgments	419
Appendix B. Manifest constants	421
Appendix C. Revision History	422
Appendix D. Notices	424

1 Introduction

This document describes the basic PKCS#11 token interface and token behavior.

The PKCS#11 standard specifies an application programming interface (API), called “Cryptoki,” for devices that hold cryptographic information and perform cryptographic functions. Cryptoki follows a simple object based approach, addressing the goals of technology independence (any kind of device) and resource sharing (multiple applications accessing multiple devices), presenting to applications a common, logical view of the device called a “cryptographic token”.

This document specifies the data types and functions available to an application requiring cryptographic services using the ANSI C programming language. The supplier of a Cryptoki library implementation typically provides these data types and functions via ANSI C header files. Generic ANSI C header files for Cryptoki are available from the PKCS#11 web page. This document and up-to-date errata for Cryptoki will also be available from the same place.

Additional documents may provide a generic, language-independent Cryptoki interface and/or bindings between Cryptoki and other programming languages.

Cryptoki isolates an application from the details of the cryptographic device. The application does not have to change to interface to a different type of device or to run in a different environment; thus, the application is portable. How Cryptoki provides this isolation is beyond the scope of this document, although some conventions for the support of multiple types of device will be addressed here and possibly in a separate document.

Details of cryptographic mechanisms (algorithms) may be found in the associated PKCS#11 Mechanisms documents.

1.1 Definitions

For the purposes of this standard, the following definitions apply:

AES	Advanced Encryption Standard, as defined in FIPS PUB 197.
API	Application programming interface.
Application	Any computer program that calls the Cryptoki interface.
ASN.1	Abstract Syntax Notation One, as defined in X.680.
Attribute	A characteristic of an object.
BER	Basic Encoding Rules, as defined in X.690.
BLOWFISH	The Blowfish Encryption Algorithm of Bruce Schneier, www.schneier.com .
CAMELLIA	The Camellia encryption algorithm, as defined in RFC 3713.
CBC	Cipher-Block Chaining mode, as defined in FIPS PUB 81.
Certificate	A signed message binding a subject name and a public key, or a subject name and a set of attributes.
CDMF	Commercial Data Masking Facility, a block encipherment method specified by International Business Machines Corporation and based on DES.
CMAC	Cipher-based Message Authenticate Code as defined in [NIST sp800-38b] and [RFC 4493].
CMS	Cryptographic Message Syntax (see RFC 5652)

42	Cryptographic Device	A device storing cryptographic information and possibly performing cryptographic functions. May be implemented as a smart card, smart disk, PCMCIA card, or with some other technology, including software-only.
46	Cryptoki	The Cryptographic Token Interface defined in this standard.
47	Cryptoki library	A library that implements the functions specified in this standard.
48	CT-KIP	Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP])
50	DER	Distinguished Encoding Rules, as defined in X.690.
51	DES	Data Encryption Standard, as defined in FIPS PUB 46-3.
52	DSA	Digital Signature Algorithm, as defined in FIPS PUB 186-4.
53	EC	Elliptic Curve
54	ECB	Electronic Codebook mode, as defined in FIPS PUB 81.
55	ECDH	Elliptic Curve Diffie-Hellman.
56	ECDSA	Elliptic Curve DSA, as in ANSI X9.62.
57	ECMQV	Elliptic Curve Menezes-Qu-Vanstone
58	GOST 28147-89	The encryption algorithm, as defined in Part 2 [GOST 28147-89] and [RFC 4357] [RFC 4490], and RFC [4491].
60	GOST R 34.11-94	Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].
62	GOST R 34.10-2001	The digital signature algorithm, as defined in [GOST R 34.10-2001] and [RFC 4357], [RFC 4490], and [RFC 4491].
64	IV	Initialization Vector.
65	MAC	Message Authentication Code.
66	Mechanism	A process for implementing a cryptographic operation.
67	MQV	Menezes-Qu-Vanstone
68	OAEP	Optimal Asymmetric Encryption Padding for RSA.
69	Object	An item that is stored on a token. May be data, a certificate, or a key.
71	PIN	Personal Identification Number.
72	PKCS	Public-Key Cryptography Standards.
73	PRF	Pseudo random function.
74	PTD	Personal Trusted Device, as defined in MeT-PTD
75	RSA	The RSA public-key cryptosystem.
76	Reader	The means by which information is exchanged with a device.
77	Session	A logical connection between an application and a token.
78	SHA-1	The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in FIPS PUB 180-2.
80	SHA-224	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.

83	SHA-256	The Secure Hash Algorithm with a 256-bit message digest, as
84		defined in FIPS PUB 180-2.
85	SHA-384	The Secure Hash Algorithm with a 384-bit message digest, as
86		defined in FIPS PUB 180-2.
87	SHA-512	The Secure Hash Algorithm with a 512-bit message digest, as
88		defined in FIPS PUB 180-2.
89	Slot	A logical reader that potentially contains a token.
90	SSL	The Secure Sockets Layer 3.0 protocol.
91	Subject Name	The X.500 distinguished name of the entity to which a key is
92		assigned.
93	SO	A Security Officer user.
94	TLS	Transport Layer Security.
95	Token	The logical view of a cryptographic device defined by Cryptoki.
96	User	The person using an application that interfaces to Cryptoki.
97	UTF-8	Universal Character Set (UCS) transformation format (UTF) that
98		represents ISO 10646 and UNICODE strings with a variable number
99		of octets.
100	WTLS	Wireless Transport Layer Security.

101 1.2 Symbols and abbreviations

102 The following symbols are used in this standard:

103 *Table 1, Symbols*

Symbol	Definition
N/A	Not applicable
R/O	Read-only
R/W	Read/write

104 The following prefixes are used in this standard:

105 *Table 2, Prefixes*

Prefix	Description
C_	Function
CK_	Data type or general constant
CKA_	Attribute
CKC_	Certificate type
CKD_	Key derivation function
CKF_	Bit flag
CKG_	Mask generation function
CKH_	Hardware feature type
CKK_	Key type
CKM_	Mechanism type
CKN_	Notification
CKO_	Object class

Prefix	Description
CKP_	Pseudo-random function
CKS_	Session state
CKR_	Return value
CKU_	User type
CKZ_	Salt/Encoding parameter source
h	a handle
ul	a CK_ULONG
p	a pointer
pb	a pointer to a CK_BYTE
ph	a pointer to a handle
pul	a pointer to a CK_ULONG

Cryptoki is based on ANSI C types, and defines the following data types:

```

/* an unsigned 8-bit value */
typedef unsigned char CK_BYTE;

/* an unsigned 8-bit character */
typedef CK_BYTE CK_CHAR;

/* an 8-bit UTF-8 character */
typedef CK_BYTE CK_UTF8CHAR;

/* a BYTE-sized Boolean flag */
typedef CK_BYTE CK_BBOOL;

/* an unsigned value, at least 32 bits long */
typedef unsigned long int CK_ULONG;

/* a signed value, the same size as a CK_ULONG */
typedef long int CK_LONG;

/* at least 32 bits; each bit is a Boolean flag */
typedef CK_ULONG CK_FLAGS;

```

Cryptoki also uses pointers to some of these data types, as well as to the type void, which are implementation-dependent. These pointer types are:

```

CK_BYTE_PTR      /* Pointer to a CK_BYTE */
CK_CHAR_PTR      /* Pointer to a CK_CHAR */
CK_UTF8CHAR_PTR  /* Pointer to a CK_UTF8CHAR */
CK_ULONG_PTR     /* Pointer to a CK_ULONG */
CK_VOID_PTR      /* Pointer to a void */

```

Cryptoki also defines a pointer to a CK_VOID_PTR, which is implementation-dependent:

```

CK_VOID_PTR_PTR  /* Pointer to a CK_VOID_PTR */

```

In addition, Cryptoki defines a C-style NULL pointer, which is distinct from any valid pointer:

```

NULL_PTR        /* A NULL pointer */

```

It follows that many of the data and pointer types will vary somewhat from one environment to another (e.g., a CK_ULONG will sometimes be 32 bits, and sometimes perhaps 64 bits). However, these details should not affect an application, assuming it is compiled with Cryptoki header files consistent with the Cryptoki library to which the application is linked.

All numbers and values expressed in this document are decimal, unless they are preceded by “0x”, in which case they are hexadecimal values.

The **CK_CHAR** data type holds characters from the following table, taken from ANSI C:

Table 3, Character Set

Category	Characters
Letters	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z
Numbers	0 1 2 3 4 5 6 7 8 9
Graphic characters	! " # % & ' () * + , - . / : ; < = > ? [\] ^ _ { } ~
Blank character	' '

The **CK_UTF8CHAR** data type holds UTF-8 encoded Unicode characters as specified in RFC2279. UTF-8 allows internationalization while maintaining backward compatibility with the Local String definition of PKCS #11 version 2.01.

In Cryptoki, the **CK_BBOOL** data type is a Boolean type that can be true or false. A zero value means false, and a nonzero value means true. Similarly, an individual bit flag, **CKF_...**, can also be set (true) or unset (false). For convenience, Cryptoki defines the following macros for use with values of type **CK_BBOOL**:

```
#define CK_FALSE 0
#define CK_TRUE 1
```

For backwards compatibility, header files for this version of Cryptoki also define TRUE and FALSE as (CK_DISABLE_TRUE_FALSE may be set by the application vendor):

```
#ifndef CK_DISABLE_TRUE_FALSE
#ifndef FALSE
#define FALSE CK_FALSE
#endif

#ifndef TRUE
#define TRUE CK_TRUE
#endif
#endif
```

1.3 Normative References

[ARIA] National Security Research Institute, Korea, “Block Cipher Algorithm ARIA”,
URL: <https://www.ietf.org/rfc/rfc5794.txt>

[BLOWFISH] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish),
December 1993.
URL: <https://www.schneier.com/paper-blowfish-fse.html>

[CAMELLIA] M. Matsui, J. Nakajima, S. Moriai. A Description of the Camellia Encryption Algorithm,
April 2004.
URL: <http://www.ietf.org/rfc/rfc3713.txt>

[CDMF] Johnson, D.B The Commercial Data Masking Facility (CDMF) data privacy algorithm, March
1994.
URL: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5389557>

186 **[CHACHA]** D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.
 187 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>

188 **[DH]** W. Diffie, M. Hellman. New Directions in Cryptography. Nov, 1976.
 189 URL: <http://www-ee.stanford.edu/~hellman/publications/24.pdf>

190 **[FIPS PUB 46-3]** NIST. *FIPS 46-3: Data Encryption Standard*. October 1999.
 191 URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>

192 **[FIPS PUB 81]** NIST. *FIPS 81: DES Modes of Operation*. December 1980.
 193 URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>

194 **[FIPS PUB 186-4]** NIST. FIPS 186-4: Digital Signature Standard. July, 2013.
 195 URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>

196 **[FIPS SP 800-56A]** NIST. Special Publication 800-56A Revision 2: *Recommendation for Pair-Wise*
 197 *Key Establishment Schemes Using Discrete Logarithm Cryptography*, May 2013.
 198 URL: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>

199 **[FIPS SP 800-108]** NIST. Special Publication 800-108 (Revised): *Recommendation for Key*
 200 *Derivation Using Pseudorandom Functions*, October 2009.
 201 URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>

202 **[GOST]** V. Dolmatov, A. Degtyarev. GOST R. 34.11-2012: Hash Function. August 2013.
 203 URL: <https://tools.ietf.org/html/rfc6986>

204 **[MD2]** B. Kaliski. RSA Laboratories. The MD2 Message-Digest Algorithm. April, 1992.
 205 URL: <https://www.ietf.org/rfc/rfc1319.txt>

206 **[MD5]** RSA Data Security. R. Rivest. The MD5 Message-Digest Algorithm. April, 1992.
 207 URL: <https://www.ietf.org/rfc/rfc1321.txt>

208 **[NIST 802-208]** NIST Special Publication 800-208: *Recommendation for Stateful Hash-Based Signature*
 209 *Schemes*, October 2020.
 210 URL: <https://csrc.nist.gov/publications/detail/sp/800-208/final>

211 **[OAEP]** M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to Encrypt with RSA. Nov 19,
 212 1995.

213 **[PKCS11-Hist]** *PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version*
 214 *3.1. Work in progress. Latest stage: <URL>*

215 **[PKCS11-Prof]** *PKCS #11 Profiles Version 3.1*. Edited by Tim Hudson. Latest stage: [https://docs.oasis-](https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html)
 216 [open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html](https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.1/pkcs11-profiles-v3.1.html).

217 **[PKCS #1]** RSA Laboratories. *RSA Cryptography Standard*. v2.1, June 14, 2002.
 218 URL: <https://tools.ietf.org/html/rfc8017>

219 **[PKCS #3]** RSA Laboratories. *Diffie-Hellman Key-Agreement Standard*. v1.4, November 1993.
 220 URL: <ftp://ftp.rsasecurity.com/pub/pkcs/doc/pkcs-3.doc>

221 **[PKCS #5]** RSA Laboratories. *Password-Based Encryption Standard*. v2.0, March 25, 1999
 222 URL: <https://tools.ietf.org/html/rfc8018>

223 **[PKCS #7]** RSA Laboratories. *Cryptographic Message Syntax Standard*. v1.5, November 1993
 224 URL: <https://tools.ietf.org/html/rfc2315>

225 **[PKCS #8]** RSA Laboratories. *Private-Key Information Syntax Standard*. v1.2, November 1993.
 226 URL: <https://tools.ietf.org/html/rfc5958>

227 **[PKCS #12]** RSA Laboratories. *Personal Information Exchange Syntax Standard*. v1.0, June 1999.
 228 URL: <https://tools.ietf.org/html/rfc7292>

229 **[POLY1305]** D.J. Bernstein. The Poly1305-AES message-authentication code. Jan 2005.
 230 URL: <https://cr.yp.to/mac/poly1305-20050329.pdf>

231 **[RFC 2409]** D. Harkins, D. Carrel. RFC 2409: *The Internet Key Exchange (IKE)*, November 1998.
 232 URL: <https://tools.ietf.org/html/rfc2409>

233 **[RFC 2119]** Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC
 234 2119, March 1997.
 235 URL: <http://www.ietf.org/rfc/rfc2119.txt>.

236 **[RFC 2279]** F. Yergeau. RFC 2279: UTF-8, a transformation format of ISO 10646. Alis Technologies,
 237 January 1998.
 238 URL: <http://www.ietf.org/rfc/rfc2279.txt>

239 **[RFC 2534]** Masinter, L., Wing, D., Mutz, A., and K. Holtman. RFC 2534: *Media Features for Display,*
 240 *Print, and Fax*. March 1999.
 241 URL: <http://www.ietf.org/rfc/rfc2534.txt>

242 **[RFC 5652]** R. Housley. RFC 5652: *Cryptographic Message Syntax*. September 2009. URL:
 243 <http://www.ietf.org/rfc/rfc5652.txt>

244 **[RFC 5707]** Rescorla, E., "The Keying Material Exporters for Transport Layer Security (TLS)", RFC
 245 5705, March 2010.
 246 URL: <http://www.ietf.org/rfc/rfc5705.txt>

247 **[RFC 5996]** C. Kaufman, P. Hoffman, Y. Nir, P. Eronen. RFC 5996: *Internet Key Exchange Protocol*
 248 *Version 2 (IKEv2)*, September 2010.
 249 URL: <https://tools.ietf.org/html/rfc5996>

250 **[RFC 8554]** D. McGrew, M. Curcio, S. Fluhrer. RFC 8554: *Leighton-Micali Hash-Based Signatures*,
 251 April 2019.
 252 URL: <https://tools.ietf.org/html/rfc8554>

253 **[RIPEMD]** H. Dobbertin, A. Bosselaers, B. Preneel. The hash function RIPEMD-160, Feb 13, 2012.
 254 URL: <http://homes.esat.kuleuven.be/~bosselae/ripemd160.html>

255 **[SALSA]** D. Bernstein, ChaCha, a variant of Salsa20, Jan 2008.
 256 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>

257 **[SEED]** KISA. SEED 128 Algorithm Specification. Sep 2003.
 258 URL: http://seed.kisa.or.kr/html/egovframework/iwt/ds/ko/ref/%5B2%5D_SEED+128_Specification_english_M.pdf
 259 [h_M.pdf](http://seed.kisa.or.kr/html/egovframework/iwt/ds/ko/ref/%5B2%5D_SEED+128_Specification_english_M.pdf)

260 **[SHA-1]** NIST. **FIPS 180-4: Secure Hash Standard. March 2012.**
 261 URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>

262 **[SHA-2]** NIST. **FIPS 180-4: Secure Hash Standard. March 2012.**
 263 URL: <http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf>

264 **[TLS]** [RFC2246] Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999.
 265 URL: <http://www.ietf.org/rfc/rfc2246.txt>, superseded by [RFC4346] Dierks, T. and E. Rescorla, "The
 266 Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006. URL:
 267 <http://www.ietf.org/rfc/rfc4346.txt>, which was superseded by [TLS12].

268 **[TLS12]** [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol
 269 Version 1.2", RFC 5246, August 2008.
 270 URL: <http://www.ietf.org/rfc/rfc5246.txt>

271 **[TWOFISH]** B. Schneier, J. Kelsey, D. Whiting, C. Hall, N. Ferguson. **Twofish: A 128-Bit Block**
 272 **Cipher. June 15, 1998.**
 273 URL: <https://www.schneier.com/academic/twofish/>

274 **[X.500]** ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of
 275 Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1

276 **[X.509]** ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key
 277 and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8

278 **[X.680]** ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic
279 Notation. July 2002. Identical to ISO/IEC 8824-1

280 **[X.690]** ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules
281 (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical
282 to ISO/IEC 8825-1

283

284 **1.4 Non-Normative References**

285 **[CAP-1.2]** *Common Alerting Protocol Version 1.2*. 01 July 2010. OASIS Standard.
286 URL: <http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html>

287 **[AES KEYWRAP]** National Institute of Standards and Technology, NIST Special Publication 800-
288 38F, Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping, December
289 2012, <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf>

290 **[ANSI C]** ANSI/ISO. American National Standard for Programming Languages – C. 1990.

291 **[ANSI X9.31]** Accredited Standards Committee X9. Digital Signatures Using Reversible Public Key
292 Cryptography for the Financial Services Industry (rDSA). 1998.

293 **[ANSI X9.42]** Accredited Standards Committee X9. Public Key Cryptography for the Financial Services
294 Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. 2003.

295 **[ANSI X9.62]** Accredited Standards Committee X9. Public Key Cryptography for the Financial Services
296 Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA). 1998.

297 **[ANSI X9.63]** Accredited Standards Committee X9. Public Key Cryptography for the Financial Services
298 Industry: Key Agreement and Key Transport Using Elliptic Curve Cryptography. 2001.
299 URL: <http://webstore.ansi.org/RecordDetail.aspx?sku=X9.63-2011>

300 **[BRAINPOOL]** ECC Brainpool Standard Curves and Curve Generation, v1.0, 19.10.2005
301 URL: <http://www.ecc-brainpool.org>

302 **[CC/PP]** W3C. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies.
303 World Wide Web Consortium, January 2004.
304 URL: <http://www.w3.org/TR/CCPP-struct-vocab/>

305 **[CDPD]** Ameritech Mobile Communications et al. Cellular Digital Packet Data System Specifications: Part
306 406: Airlink Security. 1993.

307 **[CT-KIP]** RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0,
308 December 2005.

309 **[GCS-API]** X/Open Company Ltd. Generic Cryptographic Service API (GCS-API), Base - Draft 2.
310 February 14, 1995.

311 **[ISO/IEC 7816-1]** ISO. Information Technology — Identification Cards — Integrated Circuit(s) with
312 Contacts — Part 1: Physical Characteristics. 1998.

313 **[ISO/IEC 7816-4]** ISO. Information Technology — Identification Cards — Integrated Circuit(s) with
314 Contacts — Part 4: Interindustry Commands for Interchange. 1995.

315 **[ISO/IEC 8824-1]** ISO. Information Technology-- Abstract Syntax Notation One (ASN.1):
316 Specification of Basic Notation. 2002.

317 **[ISO/IEC 8825-1]** ISO. Information Technology—ASN.1 Encoding Rules: Specification of Basic
318 Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER).
319 2002.

320 **[ISO/IEC 9594-1]** ISO. Information Technology — Open Systems Interconnection — The Directory:
321 Overview of Concepts, Models and Services. 2001.

322 **[ISO/IEC 9594-8]** ISO. Information Technology — Open Systems Interconnection — The Directory:
 323 Public-key and Attribute Certificate Frameworks. 2001

324 **[ISO/IEC 9796-2]** ISO. Information Technology — Security Techniques — Digital Signature
 325 Scheme Giving Message Recovery — Part 2: Integer factorization based mechanisms. 2002.

326 **[Java MIDP]** Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition.
 327 November 2002.
 328 URL: <http://jcp.org/jsr/detail/118.jsp>

329 **[LEGIFRANCE]** Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français (Publication of
 330 Elliptic Curve parameters by the French state)
 331 URL: <https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816>

332 **[MeT-PTD]** MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February
 333 2003.
 334 URL: <http://www.mobiletransaction.org>

335 **[NIST AES CTS]** National Institute of Standards and Technology, Addendum to NIST Special
 336 Publication 800-38A, “Recommendation for Block Cipher Modes of Operation: Three Variants of
 337 Ciphertext Stealing for CBC Mode”
 338 URL: http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf

339 **[PCMCIA]** Personal Computer Memory Card International Association. *PC Card Standard*,
 340 Release 2.1., July 1993.

341 **[RFC 2865]** Rigney et al, “Remote Authentication Dial In User Service (RADIUS)”, IETF RFC2865,
 342 June 2000.
 343 URL: <http://www.ietf.org/rfc/rfc2865.txt>.

344 **[RFC 3686]** Housley, “Using Advanced Encryption Standard (AES) Counter Mode With IPsec
 345 Encapsulating Security Payload (ESP),” IETF RFC 3686, January 2004.
 346 URL: <http://www.ietf.org/rfc/rfc3686.txt>.

347 **[RFC 3717]** Matsui, et al, “A Description of the Camellia Encryption Algorithm,” IETF RFC 3717, April
 348 2004.
 349 URL: <http://www.ietf.org/rfc/rfc3713.txt>.

350 **[RFC 3610]** Whiting, D., Housley, R., and N. Ferguson, “Counter with CBC-MAC (CCM)”, IETF RFC
 351 3610, September 2003.
 352 URL: <http://www.ietf.org/rfc/rfc3610.txt>

353 **[RFC 3874]** Smit et al, “A 224-bit One-way Hash Function: SHA-224,” IETF RFC 3874, June 2004.
 354 URL: <http://www.ietf.org/rfc/rfc3874.txt>.

355 **[RFC 3748]** Aboba et al, “Extensible Authentication Protocol (EAP)”, IETF RFC 3748, June 2004.
 356 URL: <http://www.ietf.org/rfc/rfc3748.txt>.

357 **[RFC 4269]** South Korean Information Security Agency (KISA) “The SEED Encryption Algorithm”,
 358 December 2005.
 359 URL: <https://ftp.rfc-editor.org/in-notes/rfc4269.txt>

360 **[RFC 4309]** Housley, R., “Using Advanced Encryption Standard (AES) CCM Mode with IPsec
 361 Encapsulating Security Payload (ESP),” IETF RFC 4309, December 2005.
 362 URL: <http://www.ietf.org/rfc/rfc4309.txt>

363 **[RFC 4357]** V. Popov, I. Kurepkin, S. Leontiev “Additional Cryptographic Algorithms for Use with
 364 GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms”, January
 365 2006.
 366 URL: <http://www.ietf.org/rfc/rfc4357.txt>

367 **[RFC 4490]** S. Leontiev, Ed. G. Chudov, Ed. “Using the GOST 28147-89, GOST R 34.11-94, GOST
 368 R 34.10-94, and GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)”, May 2006.
 369 URL: <http://www.ietf.org/rfc/rfc4490.txt>

370 **[RFC 4491]** S. Leontiev, Ed., D. Shefanovski, Ed., “Using the GOST R 34.10-94, GOST R 34.10-
371 2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and
372 CRL Profile”, May 2006.
373 URL: <http://www.ietf.org/rfc/rfc4491.txt>

374 **[RFC 4493]** J. Song et al. *RFC 4493: The AES-CMAC Algorithm*. June 2006.
375 URL: <http://www.ietf.org/rfc/rfc4493.txt>

376 **[RFC 5705]** Rescorla, E., “The Keying Material Exporters for Transport Layer Security (TLS)”, RFC
377 5705, March 2010.
378 URL: <http://www.ietf.org/rfc/rfc5705.txt>

379 **[RFC 5869]** H. Krawczyk, P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function
380 (HKDF)”, May 2010
381 URL: <http://www.ietf.org/rfc/rfc5869.txt>

382 **[RFC 7539]** Y Nir, A. Langley. *RFC 7539: ChaCha20 and Poly1305 for IETF Protocols*, May 2015
383 URL: <https://tools.ietf.org/rfc/rfc7539.txt>

384 **[RFC 7748]** Aboba et al, “Elliptic Curves for Security”, IETF RFC 7748, January 2016
385 URL: <https://tools.ietf.org/html/rfc7748>

386 **[RFC 8032]** Aboba et al, “Edwards-Curve Digital Signature Algorithm (EdDSA)”, IETF RFC 8032,
387 January 2017
388 URL: <https://tools.ietf.org/html/rfc8032>

389 **[SEC 1]** Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC)*
390 *1: Elliptic Curve Cryptography*. Version 1.0, September 20, 2000.

391 **[SEC 2]** Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC)*
392 *2: Recommended Elliptic Curve Domain Parameters*. Version 1.0, September 20, 2000.

393 **[WTLS]** WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.
394 URL: <http://openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf>

395 **[XEDDSA]** The XEdDSA and VEdDSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin
396 (editor)
397 URL: <https://signal.org/docs/specifications/xeddsa/>

398 **[X.500]** ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of
399 Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1

400 **[X.509]** ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key
401 and Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8

402 **[X.680]** ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic
403 Notation. July 2002. Identical to ISO/IEC 8824-1

404 **[X.690]** ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules
405 (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical
406 to ISO/IEC 8825-1

2 Platform- and compiler-dependent directives for C or C++

There is a large array of Cryptoki-related data types that are defined in the Cryptoki header files. Certain packing and pointer-related aspects of these types are platform and compiler-dependent; these aspects are therefore resolved on a platform-by-platform (or compiler-by-compiler) basis outside of the Cryptoki header files by means of preprocessor directives.

This means that when writing C or C++ code, certain preprocessor directives **MUST** be issued before including a Cryptoki header file. These directives are described in the remainder of this section.

Platform specific implementation hints can be found in the pkcs11.h header file.

2.1 Structure packing

Cryptoki structures are packed to occupy as little space as is possible. Cryptoki structures **SHALL** be packed with 1-byte alignment.

2.2 Pointer-related macros

Because different platforms and compilers have different ways of dealing with different types of pointers, the following 6 macros **SHALL** be set outside the scope of Cryptoki:

◆ CK_PTR

CK_PTR is the “indirection string” a given platform and compiler uses to make a pointer to an object. It is used in the following fashion:

```
typedef CK_BYTE CK_PTR CK_BYTE_PTR;
```

◆ CK_DECLARE_FUNCTION

CK_DECLARE_FUNCTION(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It **SHALL** be used in the following fashion:

```
CK_DECLARE_FUNCTION(CK_RV, C_Initialize)(  
    CK_VOID_PTR pReserved  
);
```

◆ CK_DECLARE_FUNCTION_POINTER

CK_DECLARE_FUNCTION_POINTER(returnType, name), when followed by a parentheses-enclosed list of arguments and a semicolon, declares a variable or type which is a pointer to a Cryptoki API function in a Cryptoki library. returnType is the return type of the function, and name is its name. It **SHALL** be used in either of the following fashions to define a function pointer variable, myC_Initialize, which can point to a C_Initialize function in a Cryptoki library (note that neither of the following code snippets actually assigns a value to myC_Initialize):

```
CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_Initialize)(  
    CK_VOID_PTR pReserved  
);
```

or:

```
typedef CK_DECLARE_FUNCTION_POINTER(CK_RV, myC_InitializeType)(
```

```
446     CK_VOID_PTR pReserved
447 );
448 myC_InitializeType myC_Initialize;
```

449 ♦ CK_CALLBACK_FUNCTION

450 CK_CALLBACK_FUNCTION(returnType, name), when followed by a parentheses-enclosed
451 list of arguments and a semicolon, declares a variable or type which is a pointer to an application callback
452 function that can be used by a Cryptoki API function in a Cryptoki library. returnType is the return type of
453 the function, and name is its name. It SHALL be used in either of the following fashions to define a
454 function pointer variable, myCallback, which can point to an application callback which takes arguments
455 args and returns a CK_RV (note that neither of the following code snippets actually assigns a value to
456 myCallback):

```
457     CK_CALLBACK_FUNCTION(CK_RV, myCallback)(args);
458
```

459 or:

```
460     typedef CK_CALLBACK_FUNCTION(CK_RV, myCallbackType)(args);
461     myCallbackType myCallback;
```

462 ♦ NULL_PTR

463 NULL_PTR is the value of a NULL pointer. In any ANSI C environment—and in many others as well—
464 NULL_PTR SHALL be defined simply as 0.

3 General data types

The general Cryptoki data types are described in the following subsections. The data types for holding parameters for various mechanisms, and the pointers to those parameters, are not described here; these types are described with the information on the mechanisms themselves, in Section 6.

A C or C++ source file in a Cryptoki application or library can define all these types (the types described here and the types that are specifically used for particular mechanism parameters) by including the top-level Cryptoki include file, `pkcs11.h`. `pkcs11.h`, in turn, includes the other Cryptoki include files, `pkcs11t.h` and `pkcs11f.h`. A source file can also include just `pkcs11t.h` (instead of `pkcs11.h`); this defines most (but not all) of the types specified here.

When including either of these header files, a source file **MUST** specify the preprocessor directives indicated in Section 2.

3.1 General information

Cryptoki represents general information with the following types:

◆ **CK_VERSION; CK_VERSION_PTR**

CK_VERSION is a structure that describes the version of a Cryptoki interface, a Cryptoki library, or an SSL or TLS implementation, or the hardware or firmware version of a slot or token. It is defined as follows:

```
typedef struct CK_VERSION {
    CK_BYTE major;
    CK_BYTE minor;
} CK_VERSION;
```

The fields of the structure have the following meanings:

major major version number (the integer portion of the version)

minor minor version number (the hundredths portion of the version)

Example: For version 1.0, *major* = 1 and *minor* = 0. For version 2.10, *major* = 2 and *minor* = 10. Table 4 below lists the major and minor version values for the officially published Cryptoki specifications.

Table 4, Major and minor version values for published Cryptoki specifications

Version	major	minor
1.0	0x01	0x00
2.01	0x02	0x01
2.10	0x02	0x0a
2.11	0x02	0x0b
2.20	0x02	0x14
2.30	0x02	0x1e
2.40	0x02	0x28
3.0	0x03	0x00

Minor revisions of the Cryptoki standard are always upwardly compatible within the same major version number.

CK_VERSION_PTR is a pointer to a **CK_VERSION**.

◆ **CK_INFO; CK_INFO_PTR**

CK_INFO provides general information about Cryptoki. It is defined as follows:

```

498 typedef struct CK_INFO {
499     CK_VERSION cryptokiVersion;
500     CK_UTF8CHAR manufacturerID[32];
501     CK_FLAGS flags;
502     CK_UTF8CHAR libraryDescription[32];
503     CK_VERSION libraryVersion;
504 } CK_INFO;
505

```

506 The fields of the structure have the following meanings:

507	<i>cryptokiVersion</i>	Cryptoki interface version number, for compatibility with future revisions of this interface
508		
509	<i>manufacturerID</i>	ID of the Cryptoki library manufacturer. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated.
510		
511	<i>flags</i>	bit flags reserved for future versions. MUST be zero for this version
512	<i>libraryDescription</i>	character-string description of the library. MUST be padded with the blank character (' '). Should <i>not</i> be null-terminated.
513		
514	<i>libraryVersion</i>	Cryptoki library version number

515 For libraries written to this document, the value of *cryptokiVersion* should match the version of this specification; the value of *libraryVersion* is the version number of the library software itself.

517 **CK_INFO_PTR** is a pointer to a **CK_INFO**.

518 ♦ **CK_NOTIFICATION**

519 **CK_NOTIFICATION** holds the types of notifications that Cryptoki provides to an application. It is defined as follows:

```

521 typedef CK_ULONG CK_NOTIFICATION;
522

```

523 For this version of Cryptoki, the following types of notifications are defined:

```

524 CKN_SURRENDER
525

```

526 The notifications have the following meanings:

527	<i>CKN_SURRENDER</i>	Cryptoki is surrendering the execution of a function executing in a session so that the application may perform other operations. After performing any desired operations, the application should indicate to Cryptoki whether to continue or cancel the function (see Section 5.21.1).
528		
529		
530		
531		

532 **3.2 Slot and token types**

533 Cryptoki represents slot and token information with the following types:

534 ♦ **CK_SLOT_ID; CK_SLOT_ID_PTR**

535 **CK_SLOT_ID** is a Cryptoki-assigned value that identifies a slot. It is defined as follows:

```

536 typedef CK_ULONG CK_SLOT_ID;
537

```

538 A list of **CK_SLOT_IDs** is returned by **C_GetSlotList**. A priori, any value of **CK_SLOT_ID** can be a valid slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a slot, however.

541 **CK_SLOT_ID_PTR** is a pointer to a **CK_SLOT_ID**.

542 ♦ **CK_SLOT_INFO; CK_SLOT_INFO_PTR**

543 **CK_SLOT_INFO** provides information about a slot. It is defined as follows:

```
544 typedef struct CK_SLOT_INFO {  
545     CK_UTF8CHAR slotDescription[64];  
546     CK_UTF8CHAR manufacturerID[32];  
547     CK_FLAGS flags;  
548     CK_VERSION hardwareVersion;  
549     CK_VERSION firmwareVersion;  
550 } CK_SLOT_INFO;  
551
```

552 The fields of the structure have the following meanings:

553	<i>slotDescription</i>	character-string description of the slot. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
554		
555	<i>manufacturerID</i>	ID of the slot manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
556		
557	<i>flags</i>	bits flags that provide capabilities of the slot. The flags are defined below
558		
559	<i>hardwareVersion</i>	version number of the slot's hardware
560	<i>firmwareVersion</i>	version number of the slot's firmware

561 The following table defines the *flags* field:

562 *Table 5, Slot Information Flags*

Bit Flag	Mask	Meaning
CKF_TOKEN_PRESENT	0x00000001	True if a token is present in the slot (e.g., a device is in the reader)
CKF_REMOVABLE_DEVICE	0x00000002	True if the reader supports removable devices
CKF_HW_SLOT	0x00000004	True if the slot is a hardware slot, as opposed to a software slot implementing a "soft token"

563 For a given slot, the value of the **CKF_REMOVABLE_DEVICE** flag *never changes*. In addition, if this flag
564 is not set for a given slot, then the **CKF_TOKEN_PRESENT** flag for that slot is *always* set. That is, if a
565 slot does not support a removable device, then that slot always has a token in it.

566 **CK_SLOT_INFO_PTR** is a pointer to a **CK_SLOT_INFO**.

567 ♦ **CK_TOKEN_INFO; CK_TOKEN_INFO_PTR**

568 **CK_TOKEN_INFO** provides information about a token. It is defined as follows:

```
569 typedef struct CK_TOKEN_INFO {  
570     CK_UTF8CHAR label[32];  
571     CK_UTF8CHAR manufacturerID[32];  
572     CK_UTF8CHAR model[16];  
573     CK_CHAR serialNumber[16];  
574     CK_FLAGS flags;  
575     CK_ULONG ulMaxSessionCount;  
576     CK_ULONG ulSessionCount;  
577
```



```

577 CK_ULONG ulMaxRwSessionCount;
578 CK_ULONG ulRwSessionCount;
579 CK_ULONG ulMaxPinLen;
580 CK_ULONG ulMinPinLen;
581 CK_ULONG ulTotalPublicMemory;
582 CK_ULONG ulFreePublicMemory;
583 CK_ULONG ulTotalPrivateMemory;
584 CK_ULONG ulFreePrivateMemory;
585 CK_VERSION hardwareVersion;
586 CK_VERSION firmwareVersion;
587 CK_CHAR utcTime[16];
588 } CK_TOKEN_INFO;
589

```

590 The fields of the structure have the following meanings:

591	<i>label</i>	application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
592		
593		
594	<i>manufacturerID</i>	ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
595		
596	<i>model</i>	model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
597		
598	<i>serialNumber</i>	character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
599		
600	<i>flags</i>	bit flags indicating capabilities and status of the device as defined below
601		
602	<i>ulMaxSessionCount</i>	maximum number of sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
603		
604		
605	<i>ulSessionCount</i>	number of sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
606		
607	<i>ulMaxRwSessionCount</i>	maximum number of read/write sessions that can be opened with the token at one time by a single application (see CK_TOKEN_INFO Note below)
608		
609		
610	<i>ulRwSessionCount</i>	number of read/write sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
611		
612	<i>ulMaxPinLen</i>	maximum length in bytes of the PIN
613	<i>ulMinPinLen</i>	minimum length in bytes of the PIN
614	<i>ulTotalPublicMemory</i>	the total amount of memory on the token in bytes in which public objects may be stored (see CK_TOKEN_INFO Note below)
615		
616	<i>ulFreePublicMemory</i>	the amount of free (unused) memory on the token in bytes for public objects (see CK_TOKEN_INFO Note below)
617		
618	<i>ulTotalPrivateMemory</i>	the total amount of memory on the token in bytes in which private objects may be stored (see CK_TOKEN_INFO Note below)
619		
620	<i>ulFreePrivateMemory</i>	the amount of free (unused) memory on the token in bytes for private objects (see CK_TOKEN_INFO Note below)
621		
622	<i>hardwareVersion</i>	version number of hardware
623	<i>firmwareVersion</i>	version number of firmware

624 *utcTime* current time as a character-string of length 16, represented in the
625 format YYYYMMDDhhmmssxx (4 characters for the year; 2
626 characters each for the month, the day, the hour, the minute, and
627 the second; and 2 additional reserved '0' characters). The value of
628 this field only makes sense for tokens equipped with a clock, as
629 indicated in the token information flags (see below)

630 The following table defines the *flags* field:

631 Table 6, Token Information Flags

Bit Flag	Mask	Meaning
CKF_RNG	0x00000001	True if the token has its own random number generator
CKF_WRITE_PROTECTED	0x00000002	True if the token is write-protected (see below)
CKF_LOGIN_REQUIRED	0x00000004	True if there are some cryptographic functions that a user MUST be logged in to perform
CKF_USER_PIN_INITIALIZED	0x00000008	True if the normal user's PIN has been initialized
CKF_RESTORE_KEY_NOT_NEEDED	0x00000020	True if a successful save of a session's cryptographic operations state <i>always</i> contains all keys needed to restore the state of the session
CKF_CLOCK_ON_TOKEN	0x00000040	True if token has its own hardware clock
CKF_PROTECTED_AUTHENTICATION_PATH	0x00000100	True if token has a "protected authentication path", whereby a user can log into the token without passing a PIN through the Cryptoki library
CKF_DUAL_CRYPTO_OPERATIONS	0x00000200	True if a single session with the token can perform dual cryptographic operations (see Section 5.14)
CKF_TOKEN_INITIALIZED	0x00000400	True if the token has been initialized using C_InitToken or an equivalent mechanism outside the scope of this standard. Calling C_InitToken when this flag is set will cause the token to be reinitialized.
CKF_SECONDARY_AUTHENTICATION	0x00000800	True if the token supports secondary authentication for private key objects. (Deprecated; new implementations MUST NOT set this flag)
CKF_USER_PIN_COUNT_LOW	0x00010000	True if an incorrect user login PIN has been entered at least once since the last successful authentication.

Bit Flag	Mask	Meaning
CKF_USER_PIN_FINAL_TRY	0x00020000	True if supplying an incorrect user PIN will cause it to become locked.
CKF_USER_PIN_LOCKED	0x00040000	True if the user PIN has been locked. User login to the token is not possible.
CKF_USER_PIN_TO_BE_CHANGED	0x00080000	True if the user PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.
CKF_SO_PIN_COUNT_LOW	0x00100000	True if an incorrect SO login PIN has been entered at least once since the last successful authentication.
CKF_SO_PIN_FINAL_TRY	0x00200000	True if supplying an incorrect SO PIN will cause it to become locked.
CKF_SO_PIN_LOCKED	0x00400000	True if the SO PIN has been locked. SO login to the token is not possible.
CKF_SO_PIN_TO_BE_CHANGED	0x00800000	True if the SO PIN value is the default value set by token initialization or manufacturing, or the PIN has been expired by the card.
CKF_ERROR_STATE	0x01000000	True if the token failed a FIPS 140-2 self-test and entered an error state.

Exactly what the **CKF_WRITE_PROTECTED** flag means is not specified in Cryptoki. An application may be unable to perform certain actions on a write-protected token; these actions can include any of the following, among others:

- Creating/modifying/deleting any object on the token.
- Creating/modifying/deleting a token object on the token.
- Changing the SO's PIN.
- Changing the normal user's PIN.

The token may change the value of the **CKF_WRITE_PROTECTED** flag depending on the session state to implement its object management policy. For instance, the token may set the **CKF_WRITE_PROTECTED** flag unless the session state is R/W SO or R/W User to implement a policy that does not allow any objects, public or private, to be created, modified, or deleted unless the user has successfully called C_Login.

The **CKF_USER_PIN_COUNT_LOW**, **CKF_USER_PIN_COUNT_LOW**, **CKF_USER_PIN_FINAL_TRY**, and **CKF_SO_PIN_FINAL_TRY** flags may always be set to false if the token does not support the functionality or will not reveal the information because of its security policy.

The **CKF_USER_PIN_TO_BE_CHANGED** and **CKF_SO_PIN_TO_BE_CHANGED** flags may always be set to false if the token does not support the functionality. If a PIN is set to the default value, or has expired, the appropriate **CKF_USER_PIN_TO_BE_CHANGED** or **CKF_SO_PIN_TO_BE_CHANGED** flag is set to true. When either of these flags are true, logging in with the corresponding PIN will succeed, but only the C_SetPIN function can be called. Calling any other function that required the user to be logged in will cause CKR_PIN_EXPIRED to be returned until C_SetPIN is called successfully.

CK_TOKEN_INFO Note: The fields ulMaxSessionCount, ulSessionCount, ulMaxRwSessionCount, ulRwSessionCount, ulTotalPublicMemory, ulFreePublicMemory, ulTotalPrivateMemory, and ulFreePrivateMemory can have the special value CK_UNAVAILABLE_INFORMATION, which means that the token and/or library is unable or unwilling to provide that information. In addition, the fields ulMaxSessionCount and ulMaxRwSessionCount can have the special value CK_EFFECTIVELY_INFINITE, which means that there is no practical limit on the number of sessions (resp. R/W sessions) an application can have open with the token.

It is important to check these fields for these special values. This is particularly true for CK_EFFECTIVELY_INFINITE, since an application seeing this value in the ulMaxSessionCount or ulMaxRwSessionCount field would otherwise conclude that it can't open any sessions with the token, which is far from being the case.

The upshot of all this is that the correct way to interpret (for example) the ulMaxSessionCount field is something along the lines of the following:

```
CK_TOKEN_INFO info;
.
.
if ((CK_LONG) info.ulMaxSessionCount
    == CK_UNAVAILABLE_INFORMATION) {
    /* Token refuses to give value of ulMaxSessionCount */
    .
    .
} else if (info.ulMaxSessionCount == CK_EFFECTIVELY_INFINITE) {
    /* Application can open as many sessions as it wants */
    .
    .
} else {
    /* ulMaxSessionCount really does contain what it should */
    .
    .
}
```

CK_TOKEN_INFO_PTR is a pointer to a CK_TOKEN_INFO.

3.3 Session types

Cryptoki represents session information with the following types:

◆ CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR

CK_SESSION_HANDLE is a Cryptoki-assigned value that identifies a session. It is defined as follows:

```
typedef CK_ULONG CK_SESSION_HANDLE;
```

Valid session handles in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
CK_INVALID_HANDLE
```

CK_SESSION_HANDLE_PTR is a pointer to a CK_SESSION_HANDLE.

◆ CK_USER_TYPE

CK_USER_TYPE holds the types of Cryptoki users described in [PKCS11-UG] and, in addition, a context-specific type described in Section 4.9. It is defined as follows:

```
typedef CK_ULONG CK_USER_TYPE;
```

For this version of Cryptoki, the following types of users are defined:

```
CKU_SO
CKU_USER
CKU_CONTEXT_SPECIFIC
```

◆ CK_STATE

CK_STATE holds the session state, as described in [PKCS11-UG]. It is defined as follows:

```
typedef CK_ULONG CK_STATE;
```

For this version of Cryptoki, the following session states are defined:

```
CKS_RO_PUBLIC_SESSION
CKS_RO_USER_FUNCTIONS
CKS_RW_PUBLIC_SESSION
CKS_RW_USER_FUNCTIONS
CKS_RW_SO_FUNCTIONS
```

◆ CK_SESSION_INFO; CK_SESSION_INFO_PTR

CK_SESSION_INFO provides information about a session. It is defined as follows:

```
typedef struct CK_SESSION_INFO {
    CK_SLOT_ID slotID;
    CK_STATE state;
    CK_FLAGS flags;
    CK_ULONG ulDeviceError;
} CK_SESSION_INFO;
```

The fields of the structure have the following meanings:

<i>slotID</i>	ID of the slot that interfaces with the token
<i>state</i>	the state of the session
<i>flags</i>	bit flags that define the type of session; the flags are defined below
<i>ulDeviceError</i>	an error code defined by the cryptographic device. Used for errors not covered by Cryptoki.

The following table defines the *flags* field:

Table 7, Session Information Flags

Bit Flag	Mask	Meaning
CKF_RW_SESSION	0x00000002	True if the session is read/write; false if the session is read-only
CKF_SERIAL_SESSION	0x00000004	This flag is provided for backward compatibility, and should always be set to true

CK_SESSION_INFO_PTR is a pointer to a **CK_SESSION_INFO**.

3.4 Object types

Cryptoki represents object information with the following types:

◆ CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR

CK_OBJECT_HANDLE is a token-specific identifier for an object. It is defined as follows:

```
typedef CK_ULONG CK_OBJECT_HANDLE;
```

When an object is created or found on a token by an application, Cryptoki assigns it an object handle for that application's sessions to use to access it. A particular object on a token does not necessarily have a handle which is fixed for the lifetime of the object; however, if a particular session can use a particular handle to access a particular object, then that session will continue to be able to use that handle to access that object as long as the session continues to exist, the object continues to exist, and the object continues to be accessible to the session.

Valid object handles in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
CK_INVALID_HANDLE
```

CK_OBJECT_HANDLE_PTR is a pointer to a **CK_OBJECT_HANDLE**.

◆ CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR

CK_OBJECT_CLASS is a value that identifies the classes (or types) of objects that Cryptoki recognizes. It is defined as follows:

```
typedef CK_ULONG CK_OBJECT_CLASS;
```

Object classes are defined with the objects that use them. The type is specified on an object through the **CKA_CLASS** attribute of the object.

Vendor defined values for this type may also be specified.

```
CKO_VENDOR_DEFINED
```

Object classes **CKO_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their object classes through the PKCS process.

CK_OBJECT_CLASS_PTR is a pointer to a **CK_OBJECT_CLASS**.

◆ CK_HW_FEATURE_TYPE

CK_HW_FEATURE_TYPE is a value that identifies a hardware feature type of a device. It is defined as follows:

```
typedef CK_ULONG CK_HW_FEATURE_TYPE;
```

Hardware feature types are defined with the objects that use them. The type is specified on an object through the **CKA_HW_FEATURE_TYPE** attribute of the object.

Vendor defined values for this type may also be specified.

```
CKH_VENDOR_DEFINED
```

Feature types **CKH_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their feature types through the PKCS process.

776 ♦ **CK_KEY_TYPE**

777 **CK_KEY_TYPE** is a value that identifies a key type. It is defined as follows:

```
778 typedef CK_ULONG CK_KEY_TYPE;  
779
```

780 Key types are defined with the objects and mechanisms that use them. The key type is specified on an object through the CKA_KEY_TYPE attribute of the object.

782 Vendor defined values for this type may also be specified.

```
783 CKK_VENDOR_DEFINED  
784
```

785 Key types **CKK_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their key types through the PKCS process.

787 ♦ **CK_CERTIFICATE_TYPE**

788 **CK_CERTIFICATE_TYPE** is a value that identifies a certificate type. It is defined as follows:

```
789 typedef CK_ULONG CK_CERTIFICATE_TYPE;  
790
```

791 Certificate types are defined with the objects and mechanisms that use them. The certificate type is specified on an object through the CKA_CERTIFICATE_TYPE attribute of the object.

793 Vendor defined values for this type may also be specified.

```
794 CKC_VENDOR_DEFINED  
795
```

796 Certificate types **CKC_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their certificate types through the PKCS process.

798 ♦ **CK_CERTIFICATE_CATEGORY**

799 **CK_CERTIFICATE_CATEGORY** is a value that identifies a certificate category. It is defined as follows:

```
800 typedef CK_ULONG CK_CERTIFICATE_CATEGORY;  
801
```

802 For this version of Cryptoki, the following certificate categories are defined:

Constant	Value	Meaning
CK_CERTIFICATE_CATEGORY_UNSPECIFIED	0x00000000UL	No category specified
CK_CERTIFICATE_CATEGORY_TOKEN_USER	0x00000001UL	Certificate belongs to owner of the token
CK_CERTIFICATE_CATEGORY_AUTHORITY	0x00000002UL	Certificate belongs to a certificate authority
CK_CERTIFICATE_CATEGORY_OTHER_ENTITY	0x00000003UL	Certificate belongs to an end entity (i.e.: not a CA)

803 ♦ **CK_ATTRIBUTE_TYPE**

804 **CK_ATTRIBUTE_TYPE** is a value that identifies an attribute type. It is defined as follows:

```
805 typedef CK_ULONG CK_ATTRIBUTE_TYPE;  
806
```

Attributes are defined with the objects and mechanisms that use them. Attributes are specified on an object as a list of type, length value items. These are often specified as an attribute template.

Vendor defined values for this type may also be specified.

```
CKA_VENDOR_DEFINED
```

Attribute types **CKA_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their attribute types through the PKCS process.

◆ CK_ATTRIBUTE; CK_ATTRIBUTE_PTR

CK_ATTRIBUTE is a structure that includes the type, value, and length of an attribute. It is defined as follows:

```
typedef struct CK_ATTRIBUTE {  
    CK_ATTRIBUTE_TYPE type;  
    CK_VOID_PTR pValue;  
    CK_ULONG ulValueLen;  
} CK_ATTRIBUTE;
```

The fields of the structure have the following meanings:

<i>type</i>	the attribute type
<i>pValue</i>	pointer to the value of the attribute
<i>ulValueLen</i>	length in bytes of the value

If an attribute has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. An array of **CK_ATTRIBUTES** is called a “template” and is used for creating, manipulating and searching for objects. The order of the attributes in a template *never* matters, even if the template contains vendor-specific attributes. Note that *pValue* is a “void” pointer, facilitating the passing of arbitrary values. Both the application and Cryptoki library **MUST** ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

The constant **CK_UNAVAILABLE_INFORMATION** is used in the *ulValueLen* field to denote an invalid or unavailable value. See **C_GetAttributeValue** for further details.

CK_ATTRIBUTE_PTR is a pointer to a **CK_ATTRIBUTE**.

◆ CK_DATE

CK_DATE is a structure that defines a date. It is defined as follows:

```
typedef struct CK_DATE {  
    CK_CHAR year[4];  
    CK_CHAR month[2];  
    CK_CHAR day[2];  
} CK_DATE;
```

The fields of the structure have the following meanings:

<i>year</i>	the year (“1900” - “9999”)
<i>month</i>	the month (“01” - “12”)
<i>day</i>	the day (“01” - “31”)

The fields hold numeric characters from the character set in Table 3, not the literal byte values.

When a Cryptoki object carries an attribute of this type, and the default value of the attribute is specified to be "empty," then Cryptoki libraries SHALL set the attribute's *ulValueLen* to 0.

Note that implementations of previous versions of Cryptoki may have used other methods to identify an "empty" attribute of type CK_DATE, and applications that needs to interoperate with these libraries therefore have to be flexible in what they accept as an empty value.

◆ CK_PROFILE_ID; CK_PROFILE_ID_PTR

CK_PROFILE_ID is an unsigned long value representing a specific token profile. It is defined as follows:

```
typedef CK_ULONG CK_PROFILE_ID;
```

Profiles are defined in the PKCS #11 Cryptographic Token Interface Profiles document. s. IDs greater than 0xffffffff may cause compatibility issues on platforms that have CK_ULONG values of 32 bits, and should be avoided.

Vendor defined values for this type may also be specified.

```
CKP_VENDOR_DEFINED
```

Profile IDs **CKP_VENDOR_DEFINED** and above are permanently reserved for token vendors. For interoperability, vendors should register their object classes through the PKCS process.

Valid Profile IDs in Cryptoki always have nonzero values. For developers' convenience, Cryptoki defines the following symbolic value:

```
CKP_INVALID_ID
```

CK_PROFILE_ID_PTR is a pointer to a **CK_PROFILE_ID**.

◆ CK_JAVA_MIDP_SECURITY_DOMAIN

CK_JAVA_MIDP_SECURITY_DOMAIN is a value that identifies the Java MIDP security domain of a certificate. It is defined as follows:

```
typedef CK_ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
```

For this version of Cryptoki, the following security domains are defined. See the Java MIDP specification for further information:

Constant	Value	Meaning
CK_SECURITY_DOMAIN_UNSPECIFIED	0x00000000UL	No domain specified
CK_SECURITY_DOMAIN_MANUFACTURER	0x00000001UL	Manufacturer protection domain
CK_SECURITY_DOMAIN_OPERATOR	0x00000002UL	Operator protection domain
CK_SECURITY_DOMAIN_THIRD_PARTY	0x00000003UL	Third party protection domain

3.5 Data types for mechanisms

Cryptoki supports the following types for describing mechanisms and parameters to them:

882 ♦ **CK_MECHANISM_TYPE; CK_MECHANISM_TYPE_PTR**

883 **CK_MECHANISM_TYPE** is a value that identifies a mechanism type. It is defined as follows:

```
884 typedef CK_ULONG CK_MECHANISM_TYPE;  
885
```

886 Mechanism types are defined with the objects and mechanism descriptions that use them.

887 Vendor defined values for this type may also be specified.

```
888 CKM_VENDOR_DEFINED  
889
```

890 Mechanism types **CKM_VENDOR_DEFINED** and above are permanently reserved for token vendors.
891 For interoperability, vendors should register their mechanism types through the PKCS process.

892 **CK_MECHANISM_TYPE_PTR** is a pointer to a **CK_MECHANISM_TYPE**.

893 ♦ **CK_MECHANISM; CK_MECHANISM_PTR**

894 **CK_MECHANISM** is a structure that specifies a particular mechanism and any parameters it requires. It
895 is defined as follows:

```
896 typedef struct CK_MECHANISM {  
897     CK_MECHANISM_TYPE mechanism;  
898     CK_VOID_PTR pParameter;  
899     CK_ULONG ulParameterLen;  
900 } CK_MECHANISM;  
901
```

902 The fields of the structure have the following meanings:

903	<i>mechanism</i>	the type of mechanism
904	<i>pParameter</i>	pointer to the parameter if required by the mechanism
905	<i>ulParameterLen</i>	length in bytes of the parameter

906 Note that *pParameter* is a “void” pointer, facilitating the passing of arbitrary values. Both the application
907 and the Cryptoki library **MUST** ensure that the pointer can be safely cast to the expected type (*i.e.*,
908 without word-alignment errors).

909 **CK_MECHANISM_PTR** is a pointer to a **CK_MECHANISM**.

910 ♦ **CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR**

911 **CK_MECHANISM_INFO** is a structure that provides information about a particular mechanism. It is
912 defined as follows:

```
913 typedef struct CK_MECHANISM_INFO {  
914     CK_ULONG ulMinKeySize;  
915     CK_ULONG ulMaxKeySize;  
916     CK_FLAGS flags;  
917 } CK_MECHANISM_INFO;  
918
```

919 The fields of the structure have the following meanings:

920	<i>ulMinKeySize</i>	the minimum size of the key for the mechanism (whether this is 921 measured in bits or in bytes is mechanism-dependent)
922	<i>ulMaxKeySize</i>	the maximum size of the key for the mechanism (whether this is 923 measured in bits or in bytes is mechanism-dependent)
924	<i>flags</i>	bit flags specifying mechanism capabilities

925 For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.

926 The following table defines the *flags* field:

927 Table 8, Mechanism Information Flags

Bit Flag	Mask	Meaning
CKF_HW	0x00000001	True if the mechanism is performed by the device; false if the mechanism is performed in software
CKF_MESSAGE_ENCRYPT	0x00000002	True if the mechanism can be used with C_MessageEncryptInit
CKF_MESSAGE_DECRYPT	0x00000004	True if the mechanism can be used with C_MessageDecryptInit
CKF_MESSAGE_SIGN	0x00000008	True if the mechanism can be used with C_MessageSignInit
CKF_MESSAGE_VERIFY	0x00000010	True if the mechanism can be used with C_MessageVerifyInit
CKF_MULTI_MESSAGE	0x00000020	True if the mechanism can be used with C_*MessageBegin . One of CKF_MESSAGE_* flag must also be set.
CKF_FIND_OBJECTS	0x00000040	This flag can be passed in as a parameter to C_SessionCancel to cancel an active object search operation. Any other use of this flag is outside the scope of this standard.
CKF_ENCRYPT	0x00000100	True if the mechanism can be used with C_EncryptInit
CKF_DECRYPT	0x00000200	True if the mechanism can be used with C_DecryptInit
CKF_DIGEST	0x00000400	True if the mechanism can be used with C_DigestInit
CKF_SIGN	0x00000800	True if the mechanism can be used with C_SignInit
CKF_SIGN_RECOVER	0x00001000	True if the mechanism can be used with C_SignRecoverInit
CKF_VERIFY	0x00002000	True if the mechanism can be used with C_VerifyInit
CKF_VERIFY_RECOVER	0x00004000	True if the mechanism can be used with C_VerifyRecoverInit
CKF_GENERATE	0x00008000	True if the mechanism can be used with C_GenerateKey
CKF_GENERATE_KEY_PAIR	0x00010000	True if the mechanism can be used with C_GenerateKeyPair
CKF_WRAP	0x00020000	True if the mechanism can be used with C_WrapKey
CKF_UNWRAP	0x00040000	True if the mechanism can be used with C_UnwrapKey
CKF_DERIVE	0x00080000	True if the mechanism can be used with C_DeriveKey

Bit Flag	Mask	Meaning
CKF_EXTENSION	0x80000000	True if there is an extension to the flags; false if no extensions. MUST be false for this version.

928 CK_MECHANISM_INFO_PTR is a pointer to a CK_MECHANISM_INFO.

929 3.6 Function types

930 Cryptoki represents information about functions with the following data types:

931 ♦ CK_RV

932 **CK_RV** is a value that identifies the return value of a Cryptoki function. It is defined as follows:

```
933 typedef CK_ULONG CK_RV;
934
```

935 Vendor defined values for this type may also be specified.

```
936 CKR_VENDOR_DEFINED
937
```

938 Section 5.1 defines the meaning of each **CK_RV** value. Return values **CKR_VENDOR_DEFINED** and
939 above are permanently reserved for token vendors. For interoperability, vendors should register their
940 return values through the PKCS process.

941 ♦ CK_NOTIFY

942 **CK_NOTIFY** is the type of a pointer to a function used by Cryptoki to perform notification callbacks. It is
943 defined as follows:

```
944 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY) (
945     CK_SESSION_HANDLE hSession,
946     CK_NOTIFICATION event,
947     CK_VOID_PTR pApplication
948 );
949
```

950 The arguments to a notification callback function have the following meanings:

951	<i>hSession</i>	The handle of the session performing the callback
952	<i>event</i>	The type of notification callback
953	<i>pApplication</i>	An application-defined value. This is the same value as was passed 954 to C_OpenSession to open the session performing the callback

955 ♦ CK_C_XXX

956 Cryptoki also defines an entire family of other function pointer types. For each function **C_XXX** in the
957 Cryptoki API (see Section 4.12 for detailed information about each of them), Cryptoki defines a type
958 **CK_C_XXX**, which is a pointer to a function with the same arguments and return value as **C_XXX** has.
959 An appropriately-set variable of type **CK_C_XXX** may be used by an application to call the Cryptoki
960 function **C_XXX**.

961 ♦ **CK_FUNCTION_LIST;**
962 **CK_FUNCTION_LIST_PTR_PTR**

CK_FUNCTION_LIST_PTR;

963 **CK_FUNCTION_LIST** is a structure which contains a Cryptoki version and a function pointer to each
964 function in the Cryptoki API. It is defined as follows:

```
965 typedef struct CK_FUNCTION_LIST {  
966     CK_VERSION version;  
967     CK_C_Initialize C_Initialize;  
968     CK_C_Finalize C_Finalize;  
969     CK_C_GetInfo C_GetInfo;  
970     CK_C_GetFunctionList C_GetFunctionList;  
971     CK_C_GetSlotList C_GetSlotList;  
972     CK_C_GetSlotInfo C_GetSlotInfo;  
973     CK_C_GetTokenInfo C_GetTokenInfo;  
974     CK_C_GetMechanismList C_GetMechanismList;  
975     CK_C_GetMechanismInfo C_GetMechanismInfo;  
976     CK_C_InitToken C_InitToken;  
977     CK_C_InitPIN C_InitPIN;  
978     CK_C_SetPIN C_SetPIN;  
979     CK_C_OpenSession C_OpenSession;  
980     CK_C_CloseSession C_CloseSession;  
981     CK_C_CloseAllSessions C_CloseAllSessions;  
982     CK_C_GetSessionInfo C_GetSessionInfo;  
983  
984     CK_C_GetOperationState C_GetOperationState;  
985     CK_C_SetOperationState C_SetOperationState;  
986     CK_C_Login C_Login;  
987     CK_C_Logout C_Logout;  
988     CK_C_CreateObject C_CreateObject;  
989     CK_C_CopyObject C_CopyObject;  
990     CK_C_DestroyObject C_DestroyObject;  
991     CK_C_GetObjectSize C_GetObjectSize;  
992     CK_C_GetAttributeValue C_GetAttributeValue;  
993     CK_C_SetAttributeValue C_SetAttributeValue;  
994     CK_C_FindObjectsInit C_FindObjectsInit;  
995     CK_C_FindObjects C_FindObjects;  
996     CK_C_FindObjectsFinal C_FindObjectsFinal;  
997     CK_C_EncryptInit C_EncryptInit;  
998     CK_C_Encrypt C_Encrypt;  
999     CK_C_EncryptUpdate C_EncryptUpdate;  
1000     CK_C_EncryptFinal C_EncryptFinal;  
1001     CK_C_DecryptInit C_DecryptInit;  
1002     CK_C_Decrypt C_Decrypt;  
1003     CK_C_DecryptUpdate C_DecryptUpdate;  
1004     CK_C_DecryptFinal C_DecryptFinal;  
1005     CK_C_DigestInit C_DigestInit;  
1006     CK_C_Digest C_Digest;  
1007     CK_C_DigestUpdate C_DigestUpdate;  
1008     CK_C_DigestKey C_DigestKey;  
1009     CK_C_DigestFinal C_DigestFinal;  
1010     CK_C_SignInit C_SignInit;  
1011     CK_C_Sign C_Sign;  
1012     CK_C_SignUpdate C_SignUpdate;  
1013     CK_C_SignFinal C_SignFinal;  
1014     CK_C_SignRecoverInit C_SignRecoverInit;  
1015     CK_C_SignRecover C_SignRecover;  
1016     CK_C_VerifyInit C_VerifyInit;  
1017     CK_C_Verify C_Verify;  
1018     CK_C_VerifyUpdate C_VerifyUpdate;  
1019     CK_C_VerifyFinal C_VerifyFinal;  
1020     CK_C_VerifyRecoverInit C_VerifyRecoverInit;  
1021     CK_C_VerifyRecover C_VerifyRecover;
```

```

1022 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1023 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1024 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1025 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1026 CK_C_GenerateKey C_GenerateKey;
1027 CK_C_GenerateKeyPair C_GenerateKeyPair;
1028 CK_C_WrapKey C_WrapKey;
1029 CK_C_UnwrapKey C_UnwrapKey;
1030 CK_C_DeriveKey C_DeriveKey;
1031 CK_C_SeedRandom C_SeedRandom;
1032 CK_C_GenerateRandom C_GenerateRandom;
1033 CK_C_GetFunctionStatus C_GetFunctionStatus;
1034 CK_C_CancelFunction C_CancelFunction;
1035 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1036 } CK_FUNCTION_LIST;
1037

```

Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section 5.2). The value that this pointer points to can be used by an application to quickly find out where the executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API MUST have an entry point defined in the Cryptoki library's **CK_FUNCTION_LIST** structure. If a particular function in the Cryptoki API is not supported by a library, then the function pointer for that function in the library's **CK_FUNCTION_LIST** structure should point to a function stub which simply returns **CKR_FUNCTION_NOT_SUPPORTED**.

In this structure 'version' is the cryptoki specification version number. The major and minor versions must be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for this version of the specification may be returned via **C_GetInterfaceList** or **C_GetInterface**.

An application may or may not be able to modify a Cryptoki library's static **CK_FUNCTION_LIST** structure. Whether or not it can, it should never attempt to do so.

PKCS #11 modules must not add new functions at the end of the **CK_FUNCTION_LIST** that are not contained within the defined structure. If a PKCS#11 module needs to define additional functions, they should be placed within a vendor defined interface returned via **C_GetInterfaceList** or **C_GetInterface**.

CK_FUNCTION_LIST_PTR is a pointer to a **CK_FUNCTION_LIST**.

CK_FUNCTION_LIST_PTR_PTR is a pointer to a **CK_FUNCTION_LIST_PTR**.

◆ **CK_FUNCTION_LIST_3_0; CK_FUNCTION_LIST_3_0_PTR; CK_FUNCTION_LIST_3_0_PTR_PTR**

CK_FUNCTION_LIST_3_0 is a structure which contains the same function pointers as in **CK_FUNCTION_LIST** and additional functions added to the end of the structure that were defined in Cryptoki version 3.0. It is defined as follows:

```

1063 typedef struct CK_FUNCTION_LIST_3_0 {
1064     CK_VERSION version;
1065     CK_C_Initialize C_Initialize;
1066     CK_C_Finalize C_Finalize;
1067     CK_C_GetInfo C_GetInfo;
1068     CK_C_GetFunctionList C_GetFunctionList;
1069     CK_C_GetSlotList C_GetSlotList;
1070     CK_C_GetSlotInfo C_GetSlotInfo;
1071     CK_C_GetTokenInfo C_GetTokenInfo;
1072     CK_C_GetMechanismList C_GetMechanismList;
1073     CK_C_GetMechanismInfo C_GetMechanismInfo;
1074     CK_C_InitToken C_InitToken;
1075     CK_C_InitPIN C_InitPIN;

```

1076 CK_C_SetPIN C_SetPIN;
1077 CK_C_OpenSession C_OpenSession;
1078 CK_C_CloseSession C_CloseSession;
1079 CK_C_CloseAllSessions C_CloseAllSessions;
1080 CK_C_GetSessionInfo C_GetSessionInfo;
1081 CK_C_GetOperationState C_GetOperationState;
1082 CK_C_SetOperationState C_SetOperationState;
1083 CK_C_Login C_Login;
1084 CK_C_Logout C_Logout;
1085 CK_C_CreateObject C_CreateObject;
1086 CK_C_CopyObject C_CopyObject;
1087 CK_C_DestroyObject C_DestroyObject;
1088 CK_C_GetObjectSize C_GetObjectSize;
1089 CK_C_GetAttributeValue C_GetAttributeValue;
1090 CK_C_SetAttributeValue C_SetAttributeValue;
1091 CK_C_FindObjectsInit C_FindObjectsInit;
1092 CK_C_FindObjects C_FindObjects;
1093 CK_C_FindObjectsFinal C_FindObjectsFinal;
1094 CK_C_EncryptInit C_EncryptInit;
1095 CK_C_Encrypt C_Encrypt;
1096 CK_C_EncryptUpdate C_EncryptUpdate;
1097 CK_C_EncryptFinal C_EncryptFinal;
1098 CK_C_DecryptInit C_DecryptInit;
1099 CK_C_Decrypt C_Decrypt;
1100 CK_C_DecryptUpdate C_DecryptUpdate;
1101 CK_C_DecryptFinal C_DecryptFinal;
1102 CK_C_DigestInit C_DigestInit;
1103 CK_C_Digest C_Digest;
1104 CK_C_DigestUpdate C_DigestUpdate;
1105 CK_C_DigestKey C_DigestKey;
1106 CK_C_DigestFinal C_DigestFinal;
1107 CK_C_SignInit C_SignInit;
1108 CK_C_Sign C_Sign;
1109 CK_C_SignUpdate C_SignUpdate;
1110 CK_C_SignFinal C_SignFinal;
1111 CK_C_SignRecoverInit C_SignRecoverInit;
1112 CK_C_SignRecover C_SignRecover;
1113 CK_C_VerifyInit C_VerifyInit;
1114 CK_C_Verify C_Verify;
1115 CK_C_VerifyUpdate C_VerifyUpdate;
1116 CK_C_VerifyFinal C_VerifyFinal;
1117 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1118 CK_C_VerifyRecover C_VerifyRecover;
1119 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1120 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1121 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1122 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1123 CK_C_GenerateKey C_GenerateKey;
1124 CK_C_GenerateKeyPair C_GenerateKeyPair;
1125 CK_C_WrapKey C_WrapKey;
1126 CK_C_UnwrapKey C_UnwrapKey;
1127 CK_C_DeriveKey C_DeriveKey;
1128 CK_C_SeedRandom C_SeedRandom;
1129 CK_C_GenerateRandom C_GenerateRandom;
1130 CK_C_GetFunctionStatus C_GetFunctionStatus;
1131 CK_C_CancelFunction C_CancelFunction;
1132 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1133 CK_C_GetInterfaceList C_GetInterfaceList;
1134 CK_C_GetInterface C_GetInterface;
1135 CK_C_LoginUser C_LoginUser;
1136 CK_C_SessionCancel C_SessionCancel;
1137 CK_C_MessageEncryptInit C_MessageEncryptInit;
1138 CK_C_EncryptMessage C_EncryptMessage;
1139 CK_C_EncryptMessageBegin C_EncryptMessageBegin;

```

1140 CK_C_EncryptMessageNext C_EncryptMessageNext;
1141 CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1142 CK_C_MessageDecryptInit C_MessageDecryptInit;
1143 CK_C_DecryptMessage C_DecryptMessage;
1144 CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1145 CK_C_DecryptMessageNext C_DecryptMessageNext;
1146 CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1147 CK_C_MessageSignInit C_MessageSignInit;
1148 CK_C_SignMessage C_SignMessage;
1149 CK_C_SignMessageBegin C_SignMessageBegin;
1150 CK_C_SignMessageNext C_SignMessageNext;
1151 CK_C_MessageSignFinal C_MessageSignFinal;
1152 CK_C_MessageVerifyInit C_MessageVerifyInit;
1153 CK_C_VerifyMessage C_VerifyMessage;
1154 CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1155 CK_C_VerifyMessageNext C_VerifyMessageNext;
1156 CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1157 } CK_FUNCTION_LIST_3_0;
1158

```

1159 For a general description of **CK_FUNCTION_LIST_3_0** see **CK_FUNCTION_LIST**.

1160 In this structure, *version* is the cryptoki specification version number. It should match the value of *cryptokiVersion* returned in the **CK_INFO** structure, but must be 3.0 at minimum.

1161 This function list may be returned via **C_GetInterfaceList** or **C_GetInterface**

1162 **CK_FUNCTION_LIST_3_0_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0**.

1163 **CK_FUNCTION_LIST_3_0_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0_PTR**.

1165 ♦ **CK_INTERFACE; CK_INTERFACE_PTR; CK_INTERFACE_PTR_PTR**

1166 **CK_INTERFACE** is a structure which contains an interface name with a function list and flag.

1167 It is defined as follows:

```

1168 typedef struct CK_INTERFACE {
1169     CK_UTF8CHAR_PTR pInterfaceName;
1170     CK_VOID_PTR      pFunctionList;
1171     CK_FLAGS         flags;
1172 } CK_INTERFACE;

```

1173

1174 The fields of the structure have the following meanings:

1175	<i>pInterfaceName</i>	the name of the interface
1176	<i>pFunctionList</i>	the interface function list which must always begin with a
1177		CK_VERSION structure as the first field
1178	<i>flags</i>	bit flags specifying interface capabilities

1179 The interface name “PKCS 11” is reserved for use by interfaces defined within the cryptoki specification.

1180 Interfaces starting with the string: “Vendor ” are reserved for vendor use and will not otherwise be

1181 defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface

1182 names of “Vendor {vendor name}”. For example “Vendor ACME Inc”.

1183

1184 The following table defines the flags field:

1185 *Table 9, CK_INTERFACE Flags*

Bit Flag	Mask	Meaning
CKF_INTERFACE_FORK_SAFE	0x00000001	The returned interface will have fork tolerant semantics. When the application forks, each process will get its own copy of all session objects, session states, login states, and encryption states. Each process will also maintain access to token objects with their previously supplied handles.

CK_INTERFACE_PTR is a pointer to a **CK_INTERFACE**.

CK_INTERFACE_PTR_PTR is a pointer to a **CK_INTERFACE_PTR**.

3.7 Locking-related types

The types in this section are provided solely for applications which need to access Cryptoki from multiple threads simultaneously. *Applications which will not do this need not use any of these types.*

◆ CK_CREATEMUTEX

CK_CREATEMUTEX is the type of a pointer to an application-supplied function which creates a new mutex object and returns a pointer to it. It is defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_CREATEMUTEX) (
    CK_VOID_PTR_PTR ppMutex
);
```

Calling a **CK_CREATEMUTEX** function returns the pointer to the new mutex object in the location pointed to by **ppMutex**. Such a function should return one of the following values:

```
CKR_OK, CKR_GENERAL_ERROR
CKR_HOST_MEMORY
```

◆ CK_DESTROYMUTEX

CK_DESTROYMUTEX is the type of a pointer to an application-supplied function which destroys an existing mutex object. It is defined as follows:

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_DESTROYMUTEX) (
    CK_VOID_PTR pMutex
);
```

The argument to a **CK_DESTROYMUTEX** function is a pointer to the mutex object to be destroyed. Such a function should return one of the following values:

```
CKR_OK, CKR_GENERAL_ERROR
CKR_HOST_MEMORY
CKR_MUTEX_BAD
```

1215 ♦ **CK_LOCKMUTEX and CK_UNLOCKMUTEX**

1216 **CK_LOCKMUTEX** is the type of a pointer to an application-supplied function which locks an existing
1217 mutex object. **CK_UNLOCKMUTEX** is the type of a pointer to an application-supplied function which
1218 unlocks an existing mutex object. The proper behavior for these types of functions is as follows:

- 1219 • If a **CK_LOCKMUTEX** function is called on a mutex which is not locked, the calling thread obtains a
1220 lock on that mutex and returns.
- 1221 • If a **CK_LOCKMUTEX** function is called on a mutex which is locked by some thread other than the
1222 calling thread, the calling thread blocks and waits for that mutex to be unlocked.
- 1223 • If a **CK_LOCKMUTEX** function is called on a mutex which is locked by the calling thread, the
1224 behavior of the function call is undefined.
- 1225 • If a **CK_UNLOCKMUTEX** function is called on a mutex which is locked by the calling thread, that
1226 mutex is unlocked and the function call returns. Furthermore:
 - 1227 ○ If exactly one thread was blocking on that particular mutex, then that thread stops blocking,
1228 obtains a lock on that mutex, and its **CK_LOCKMUTEX** call returns.
 - 1229 ○ If more than one thread was blocking on that particular mutex, then exactly one of the
1230 blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on
1231 the mutex, and its **CK_LOCKMUTEX** call returns. All other threads blocking on that particular
1232 mutex continue to block.
- 1233 • If a **CK_UNLOCKMUTEX** function is called on a mutex which is not locked, then the function call
1234 returns the error code **CKR_MUTEX_NOT_LOCKED**.
- 1235 • If a **CK_UNLOCKMUTEX** function is called on a mutex which is locked by some thread other than the
1236 calling thread, the behavior of the function call is undefined.

1237 **CK_LOCKMUTEX** is defined as follows:

```
1238 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_LOCKMUTEX) (  
1239     CK_VOID_PTR pMutex  
1240 );  
1241
```

1242 The argument to a **CK_LOCKMUTEX** function is a pointer to the mutex object to be locked. Such a
1243 function should return one of the following values:

```
1244 CKR_OK, CKR_GENERAL_ERROR  
1245 CKR_HOST_MEMORY,  
1246 CKR_MUTEX_BAD  
1247
```

1248 **CK_UNLOCKMUTEX** is defined as follows:

```
1249 typedef CK_CALLBACK_FUNCTION(CK_RV, CK_UNLOCKMUTEX) (  
1250     CK_VOID_PTR pMutex  
1251 );  
1252
```

1253 The argument to a **CK_UNLOCKMUTEX** function is a pointer to the mutex object to be unlocked. Such a
1254 function should return one of the following values:

```
1255 CKR_OK, CKR_GENERAL_ERROR  
1256 CKR_HOST_MEMORY  
1257 CKR_MUTEX_BAD  
1258 CKR_MUTEX_NOT_LOCKED
```

1259 ♦ **CK_C_INITIALIZE_ARGS; CK_C_INITIALIZE_ARGS_PTR**

1260 **CK_C_INITIALIZE_ARGS** is a structure containing the optional arguments for the **C_Initialize** function.
1261 For this version of Cryptoki, these optional arguments are all concerned with the way the library deals
1262 with threads. **CK_C_INITIALIZE_ARGS** is defined as follows:

```
1263     typedef struct CK_C_INITIALIZE_ARGS {  
1264         CK_CREATEMUTEX CreateMutex;  
1265         CK_DESTROYMUTEX DestroyMutex;  
1266         CK_LOCKMUTEX LockMutex;  
1267         CK_UNLOCKMUTEX UnlockMutex;  
1268         CK_FLAGS flags;  
1269         CK_VOID_PTR pReserved;  
1270     } CK_C_INITIALIZE_ARGS;  
1271
```

1272 The fields of the structure have the following meanings:

- 1273 *CreateMutex* pointer to a function to use for creating mutex objects
- 1274 *DestroyMutex* pointer to a function to use for destroying mutex objects
- 1275 *LockMutex* pointer to a function to use for locking mutex objects
- 1276 *UnlockMutex* pointer to a function to use for unlocking mutex objects
- 1277 *flags* bit flags specifying options for **C_Initialize**; the flags are defined
1278 below
- 1279 *pReserved* reserved for future use. Should be NULL_PTR for this version of
1280 Cryptoki

1281 The following table defines the flags field:

1282 Table 10, *C_Initialize Parameter Flags*

Bit Flag	Mask	Meaning
CKF_LIBRARY_CANT_CREATE_OS_THREADS	0x00000001	True if application threads which are executing calls to the library may <i>not</i> use native operating system calls to spawn new threads; false if they may
CKF_OS_LOCKING_OK	0x00000002	True if the library can use the native operation system threading model for locking; false otherwise

1283 **CK_C_INITIALIZE_ARGS_PTR** is a pointer to a **CK_C_INITIALIZE_ARGS**.

4 Objects

Cryptoki recognizes a number of classes of objects, as defined in the **CK_OBJECT_CLASS** data type. An object consists of a set of attributes, each of which has a given value. Each attribute that an object possesses has precisely one value. The following figure illustrates the high-level hierarchy of the Cryptoki objects and some of the attributes they support:

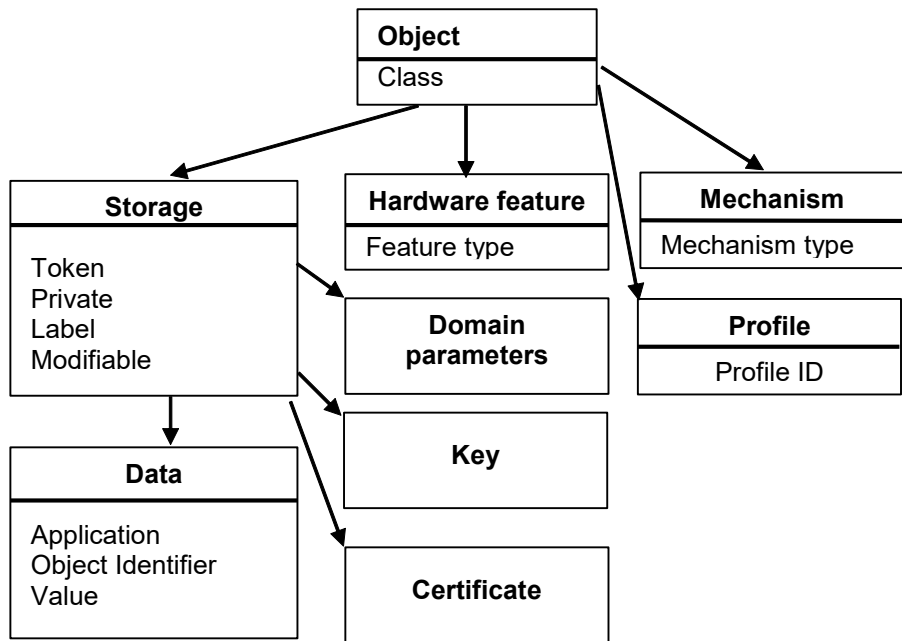


Figure 1, Object Attribute Hierarchy

Cryptoki provides functions for creating, destroying, and copying objects in general, and for obtaining and modifying the values of their attributes. Some of the cryptographic functions (e.g., **C_GenerateKey**) also create key objects to hold their results.

Objects are always “well-formed” in Cryptoki—that is, an object always contains all required attributes, and the attributes are always consistent with one another from the time the object is created. This contrasts with some object-based paradigms where an object has no attributes other than perhaps a class when it is created, and is uninitialized for some time. In Cryptoki, objects are always initialized.

Tables throughout most of Section 4 define each Cryptoki attribute in terms of the data type of the attribute value and the meaning of the attribute, which may include a default initial value. Some of the data types are defined explicitly by Cryptoki (e.g., **CK_OBJECT_CLASS**). Attribute values may also take the following types:

Byte array	an arbitrary string (array) of CK_BYTES
Big integer	a string of CK_BYTES representing an unsigned integer of arbitrary size, most-significant byte first (e.g., the integer 32768 is represented as the 2-byte string 0x80 0x00)
Local string	an unpadded string of CK_CHARS (see Table 3) with no null-termination
RFC2279 string	an unpadded string of CK_UTF8CHARs with no null-termination

A token can hold several identical objects, *i.e.*, it is permissible for two or more objects to have exactly the same values for all their attributes.

In most cases each type of object in the Cryptoki specification possesses a completely well-defined set of Cryptoki attributes. Some of these attributes possess default values, and need not be specified when creating an object; some of these default values may even be the empty string (""). Nonetheless, the object possesses these attributes. A given object has a single value for each attribute it possesses, even if the attribute is a vendor-specific attribute whose meaning is outside the scope of Cryptoki.

In addition to possessing Cryptoki attributes, objects may possess additional vendor-specific attributes whose meanings and values are not specified by Cryptoki.

4.1 Creating, modifying, and copying objects

All Cryptoki functions that create, modify, or copy objects take a template as one of their arguments, where the template specifies attribute values. Cryptographic functions that create objects (see Section 5.18) may also contribute some additional attribute values themselves; which attributes have values contributed by a cryptographic function call depends on which cryptographic mechanism is being performed (see [PKCS11-Curr] and [PKCS11-Hist] for specification of mechanisms for PKCS #11). In any case, all the required attributes supported by an object class that do not have default values **MUST** be specified when an object is created, either in the template or by the function itself.

4.1.1 Creating objects

Objects may be created with the Cryptoki functions **C_CreateObject** (see Section 5.7), **C_GenerateKey**, **C_GenerateKeyPair**, **C_UnwrapKey**, and **C_DeriveKey** (see Section 5.18). In addition, copying an existing object (with the function **C_CopyObject**) also creates a new object, but we consider this type of object creation separately in Section 4.1.3.

Attempting to create an object with any of these functions requires an appropriate template to be supplied.

1. If the supplied template specifies a value for an invalid attribute, then the attempt should fail with the error code **CKR_ATTRIBUTE_TYPE_INVALID**. An attribute is valid if it is either one of the attributes described in the Cryptoki specification or an additional vendor-specific attribute supported by the library and token.
2. If the supplied template specifies an invalid value for a valid attribute, then the attempt should fail with the error code **CKR_ATTRIBUTE_VALUE_INVALID**. The valid values for Cryptoki attributes are described in the Cryptoki specification.
3. If the supplied template specifies a value for a read-only attribute, then the attempt should fail with the error code **CKR_ATTRIBUTE_READ_ONLY**. Whether or not a given Cryptoki attribute is read-only is explicitly stated in the Cryptoki specification; however, a particular library and token may be even more restrictive than Cryptoki specifies. In other words, an attribute which Cryptoki says is not read-only may nonetheless be read-only under certain circumstances (*i.e.*, in conjunction with some combinations of other attributes) for a particular library and token. Whether or not a given non-Cryptoki attribute is read-only is obviously outside the scope of Cryptoki.
4. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are insufficient to fully specify the object to create, then the attempt should fail with the error code **CKR_TEMPLATE_INCOMPLETE**.
5. If the attribute values in the supplied template, together with any default attribute values and any attribute values contributed to the object by the object-creation function itself, are inconsistent, then the attempt should fail with the error code **CKR_TEMPLATE_INCONSISTENT**. A set of attribute values is inconsistent if not all of its members can be satisfied simultaneously *by the token*, although each value individually is valid in Cryptoki. One example of an inconsistent template would be using a template which specifies two different values for the same attribute. Another example would be trying to create a secret key object with an attribute which is appropriate for various types of public keys or private keys, but not for secret keys. A final example would be a template with an attribute that violates some token

specific requirement. Note that this final example of an inconsistent template is token-dependent—on a different token, such a template might *not* be inconsistent.

6. If the supplied template specifies the same value for a particular attribute more than once (or the template specifies the same value for a particular attribute that the object-creation function itself contributes to the object), then the behavior of Cryptoki is not completely specified. The attempt to create an object can either succeed—thereby creating the same object that would have been created if the multiply-specified attribute had only appeared once—or it can fail with error code CKR_TEMPLATE_INCONSISTENT. Library developers are encouraged to make their libraries behave as though the attribute had only appeared once in the template; application developers are strongly encouraged never to put a particular attribute into a particular template more than once.

If more than one of the situations listed above applies to an attempt to create an object, then the error code returned from the attempt can be any of the error codes from above that applies.

4.1.2 Modifying objects

Objects may be modified with the Cryptoki function **C_SetAttributeValue** (see Section 5.7). The template supplied to **C_SetAttributeValue** can contain new values for attributes which the object already possesses; values for attributes which the object does not yet possess; or both.

Some attributes of an object may be modified after the object has been created, and some may not. In addition, attributes which Cryptoki specifies are modifiable may actually *not* be modifiable on some tokens. That is, if a Cryptoki attribute is described as being modifiable, that really means only that it is modifiable *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable might depend on the values of certain attributes of the object. For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE, but not the other way around.

All the scenarios in Section 4.1.1—and the error codes they return—apply to modifying objects with **C_SetAttributeValue**, except for the possibility of a template being incomplete.

4.1.3 Copying objects

Unless an object's CKA_COPYABLE (see Table 17) attribute is set to CK_FALSE, it may be copied with the Cryptoki function **C_CopyObject** (see Section 5.7). In the process of copying an object, **C_CopyObject** also modifies the attributes of the newly-created copy according to an application-supplied template.

The Cryptoki attributes which can be modified during the course of a **C_CopyObject** operation are the same as the Cryptoki attributes which are described as being modifiable, plus the four special attributes **CKA_TOKEN**, **CKA_PRIVATE**, **CKA_MODIFIABLE** and **CKA_DESTROYABLE**. To be more precise, these attributes are modifiable during the course of a **C_CopyObject** operation *insofar as the Cryptoki specification is concerned*. A particular token might not actually support modification of some such attributes during the course of a **C_CopyObject** operation. Furthermore, whether or not a particular attribute of an object on a particular token is modifiable during the course of a **C_CopyObject** operation might depend on the values of certain attributes of the object. For example, a secret key object's **CKA_SENSITIVE** attribute can be changed from CK_FALSE to CK_TRUE during the course of a **C_CopyObject** operation, but not the other way around.

If the CKA_COPYABLE attribute of the object to be copied is set to CK_FALSE, **C_CopyObject** returns CKR_ACTION_PROHIBITED. Otherwise, the scenarios described in 10.1.1 - and the error codes they return - apply to copying objects with **C_CopyObject**, except for the possibility of a template being incomplete.

4.2 Common attributes

Table 11, Common footnotes for object attribute tables

- ¹ MUST be specified when object is created with **C_CreateObject**.
- ² MUST *not* be specified when object is created with **C_CreateObject**.
- ³ MUST be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
- ⁴ MUST *not* be specified when object is generated with **C_GenerateKey** or **C_GenerateKeyPair**.
- ⁵ MUST be specified when object is unwrapped with **C_UnwrapKey**.
- ⁶ MUST *not* be specified when object is unwrapped with **C_UnwrapKey**.
- ⁷ Cannot be revealed if object has its **CKA_SENSITIVE** attribute set to CK_TRUE or its **CKA_EXTRACTABLE** attribute set to CK_FALSE.
- ⁸ May be modified after object is created with a **C_SetAttributeValue** call, or in the process of copying object with a **C_CopyObject** call. However, it is possible that a particular token may not permit modification of the attribute during the course of a **C_CopyObject** call.
- ⁹ Default value is token-specific, and may depend on the values of other attributes.
- ¹⁰ Can only be set to CK_TRUE by the SO user.
- ¹¹ Attribute cannot be changed once set to CK_TRUE. It becomes a read only attribute.
- ¹² Attribute cannot be changed once set to CK_FALSE. It becomes a read only attribute.

1406

1407 *Table 12, Common Object Attributes*

Attribute	Data Type	Meaning
CKA_CLASS ¹	CK_OBJECT_CLASS	Object class (type)

1408 Refer to Table 11 for footnotes

1409 The above table defines the attributes common to all objects.

1410 4.3 Hardware Feature Objects

1411 4.3.1 Definitions

1412 This section defines the object class CKO_HW_FEATURE for type CK_OBJECT_CLASS as used in the
 1413 CKA_CLASS attribute of objects.

1414 4.3.2 Overview

1415 Hardware feature objects (**CKO_HW_FEATURE**) represent features of the device. They provide an easily
 1416 expandable method for introducing new value-based features to the Cryptoki interface.

1417 When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, hardware feature objects are
 1418 not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_HW_FEATURE**. This
 1419 protects applications written to previous versions of Cryptoki from finding objects that they do not
 1420 understand.

1421 *Table 13, Hardware Feature Common Attributes*

Attribute	Data Type	Meaning
CKA_HW_FEATURE_TYPE ¹	CK_HW_FEATURE_TYPE	Hardware feature (type)

1422 Refer to Table 11 for footnotes

4.3.3 Clock

4.3.3.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_CLOCK of type CK_HW_FEATURE_TYPE.

4.3.3.2 Description

Clock objects represent real-time clocks that exist on the device. This represents the same clock source as the **utcTime** field in the **CK_TOKEN_INFO** structure.

Table 14, Clock Object Attributes

Attribute	Data Type	Meaning
CKA_VALUE	CK_CHAR[16]	Current time as a character-string of length 16, represented in the format YYYYMMDDhhmmssxx (4 characters for the year; 2 characters each for the month, the day, the hour, the minute, and the second; and 2 additional reserved '0' characters).

The **CKA_VALUE** attribute may be set using the **C_SetAttributeValue** function if permitted by the device. The session used to set the time MUST be logged in. The device may require the SO to be the user logged in to modify the time value. **C_SetAttributeValue** will return the error CKR_USER_NOT_LOGGED_IN to indicate that a different user type is required to set the value.

4.3.4 Monotonic Counter Objects

4.3.4.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_MONOTONIC_COUNTER of type CK_HW_FEATURE_TYPE.

4.3.4.2 Description

Monotonic counter objects represent hardware counters that exist on the device. The counter is guaranteed to increase each time its value is read, but not necessarily by one. This might be used by an application for generating serial numbers to get some assurance of uniqueness per token.

Table 15, Monotonic Counter Attributes

Attribute	Data Type	Meaning
CKA_RESET_ON_INIT ¹	CK_BBOOL	The value of the counter will reset to a previously returned value if the token is initialized using C_InitToken .
CKA_HAS_RESET ¹	CK_BBOOL	The value of the counter has been reset at least once at some point in time.
CKA_VALUE ¹	Byte Array	The current version of the monotonic counter. The value is returned in big endian order.

¹Read Only

The **CKA_VALUE** attribute may not be set by the client.

4.3.5 User Interface Objects

4.3.5.1 Definition

The CKA_HW_FEATURE_TYPE attribute takes the value CKH_USER_INTERFACE of type CK_HW_FEATURE_TYPE.

4.3.5.2 Description

User interface objects represent the presentation capabilities of the device.

Table 16, User Interface Object Attributes

Attribute	Data type	Meaning
CKA_PIXEL_X	CK_ULONG	Screen resolution (in pixels) in X-axis (e.g. 1280)
CKA_PIXEL_Y	CK_ULONG	Screen resolution (in pixels) in Y-axis (e.g. 1024)
CKA_RESOLUTION	CK_ULONG	DPI, pixels per inch
CKA_CHAR_ROWS	CK_ULONG	For character-oriented displays; number of character rows (e.g. 24)
CKA_CHAR_COLUMNS	CK_ULONG	For character-oriented displays: number of character columns (e.g. 80). If display is of proportional-font type, this is the width of the display in "em"-s (letter "M"), see CC/PP Struct.
CKA_COLOR	CK_BBOOL	Color support
CKA_BITS_PER_PIXEL	CK_ULONG	The number of bits of color or grayscale information per pixel.
CKA_CHAR_SETS	RFC 2279 string	String indicating supported character sets, as defined by IANA MIBenum sets (www.iana.org). Supported character sets are separated with ";". E.g. a token supporting iso-8859-1 and US-ASCII would set the attribute value to "4;3".
CKA_ENCODING_METHODS	RFC 2279 string	String indicating supported content transfer encoding methods, as defined by IANA (www.iana.org). Supported methods are separated with ";". E.g. a token supporting 7bit, 8bit and base64 could set the attribute value to "7bit;8bit;base64".
CKA_MIME_TYPES	RFC 2279 string	String indicating supported (presentable) MIME-types, as defined by IANA (www.iana.org). Supported types are separated with ";". E.g. a token supporting MIME types "a/b", "a/c" and "a/d" would set the attribute value to "a/b;a/c;a/d".

The selection of attributes, and associated data types, has been done in an attempt to stay as aligned with RFC 2534 and CC/PP Struct as possible. The special value CK_UNAVAILABLE_INFORMATION may be used for CK_ULONG-based attributes when information is not available or applicable.

None of the attribute values may be set by an application.

The value of the **CKA_ENCODING_METHODS** attribute may be used when the application needs to send MIME objects with encoded content to the token.

4.4 Storage Objects

This is not an object class; hence no CKO_ definition is required. It is a category of object classes with common attributes for the object classes that follow.

Attribute	Data Type	Meaning
CKA_TOKEN	CK_BBOOL	CK_TRUE if object is a token object; CK_FALSE if object is a session object. Default is CK_FALSE.
CKA_PRIVATE	CK_BBOOL	CK_TRUE if object is a private object; CK_FALSE if object is a public object. Default value is token-specific, and may depend on the values of other attributes of the object.
CKA_MODIFIABLE	CK_BBOOL	CK_TRUE if object can be modified. Default is CK_TRUE.
CKA_LABEL	RFC2279 string	Description of the object (default empty).
CKA_COPYABLE	CK_BBOOL	CK_TRUE if object can be copied using C_CopyObject. Defaults to CK_TRUE. Can't be set to TRUE once it is set to FALSE.
CKA_DESTROYABLE	CK_BBOOL	CK_TRUE if the object can be destroyed using C_DestroyObject. Default is CK_TRUE.
CKA_UNIQUE_ID ²⁴⁶	RFC2279 string	The unique identifier assigned to the object.

1463 Only the **CKA_LABEL** attribute can be modified after the object is created. (The **CKA_TOKEN**,
 1464 **CKA_PRIVATE**, and **CKA_MODIFIABLE** attributes can be changed in the process of copying an object,
 1465 however.)

1466 The **CKA_TOKEN** attribute identifies whether the object is a token object or a session object.

1467 When the **CKA_PRIVATE** attribute is CK_TRUE, a user may not access the object until the user has
 1468 been authenticated to the token.

1469 The value of the **CKA_MODIFIABLE** attribute determines whether or not an object is read-only.

1470 The **CKA_LABEL** attribute is intended to assist users in browsing.

1471 The value of the CKA_COPYABLE attribute determines whether or not an object can be copied. This
 1472 attribute can be used in conjunction with CKA_MODIFIABLE to prevent changes to the permitted usages
 1473 of keys and other objects.

1474 The value of the CKA_DESTROYABLE attribute determines whether the object can be destroyed using
 1475 C_DestroyObject.

1476 4.4.1 The CKA_UNIQUE_ID attribute

1477 Any time a new object is created, a value for CKA_UNIQUE_ID MUST be generated by the token and
 1478 stored with the object. The specific algorithm used to generate unique ID values for objects is token-
 1479 specific, but values generated MUST be unique across all objects visible to any particular session, and
 1480 SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling
 1481 C_InitToken, MAY cause reuse of CKA_UNIQUE_ID values.

1482 Any attempt to modify the CKA_UNIQUE_ID attribute of an existing object or to specify the value of the
 1483 CKA_UNIQUE_ID attribute in the template for an operation that creates one or more objects MUST fail.
 1484 Operations failing for this reason return the error code CKR_ATTRIBUTE_READ_ONLY.

1485

4.5 Data objects

4.5.1 Definitions

This section defines the object class **CKO_DATA** for type **CK_OBJECT_CLASS** as used in the **CKA_CLASS** attribute of objects.

4.5.2 Overview

Data objects (object class **CKO_DATA**) hold information defined by an application. Other than providing access to it, Cryptoki does not attach any special meaning to a data object. The following table lists the attributes supported by data objects, in addition to the common attributes defined for this object class:

Table 18, Data Object Attributes

Attribute	Data type	Meaning
CKA_APPLICATION	RFC2279 string	Description of the application that manages the object (default empty)
CKA_OBJECT_ID	Byte Array	DER-encoding of the object identifier indicating the data object type (default empty)
CKA_VALUE	Byte array	Value of the object (default empty)

The **CKA_APPLICATION** attribute provides a means for applications to indicate ownership of the data objects they manage. Cryptoki does not provide a means of ensuring that only a particular application has access to a data object, however.

The **CKA_OBJECT_ID** attribute provides an application independent and expandable way to indicate the type of the data object value. Cryptoki does not provide a means of insuring that the data object identifier matches the data value.

The following is a sample template containing attributes for creating a data object:

```
CK_OBJECT_CLASS class = CKO_DATA;
CK_UTF8CHAR label[] = "A data object";
CK_UTF8CHAR application[] = "An application";
CK_BYTE data[] = "Sample data";
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_APPLICATION, application, sizeof(application)-1},
    {CKA_VALUE, data, sizeof(data)}
};
```

4.6 Certificate objects

4.6.1 Definitions

This section defines the object class **CKO_CERTIFICATE** for type **CK_OBJECT_CLASS** as used in the **CKA_CLASS** attribute of objects.

4.6.2 Overview

Certificate objects (object class **CKO_CERTIFICATE**) hold public-key or attribute certificates. Other than providing access to certificate objects, Cryptoki does not attach any special meaning to certificates. The following table defines the common certificate object attributes, in addition to the common attributes defined for this object class:

Table 19, Common Certificate Object Attributes

Attribute	Data type	Meaning
CKA_CERTIFICATE_TYPE ₁	CK_CERTIFICATE_TYPE	Type of certificate
CKA_TRUSTED ¹⁰	CK_BBOOL	The certificate can be trusted for the application that it was created.
CKA_CERTIFICATE_CATEGORY	CKA_CERTIFICATE_CATEGORY	(default CK_CERTIFICATE_CATEGORY_UNSPECIFIED)
CKA_CHECK_VALUE	Byte array	Checksum
CKA_START_DATE	CK_DATE	Start date for the certificate (default empty)
CKA_END_DATE	CK_DATE	End date for the certificate (default empty)
CKA_PUBLIC_KEY_INFO	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the public key contained in this certificate (default empty)

Refer to Table 11 for footnotes

Cryptoki does not enforce the relationship of the CKA_PUBLIC_KEY_INFO to the public key in the certificate, but does recommend that the key be extracted from the certificate to create this value.

The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created. This version of Cryptoki supports the following certificate types:

- X.509 public key certificate
- WTLS public key certificate
- X.509 attribute certificate

The **CKA_TRUSTED** attribute cannot be set to CK_TRUE by an application. It MUST be set by a token initialization application or by the token's SO. Trusted certificates cannot be modified.

The **CKA_CERTIFICATE_CATEGORY** attribute is used to indicate if a stored certificate is a user certificate for which the corresponding private key is available on the token ("token user"), a CA certificate ("authority"), or another end-entity certificate ("other entity"). This attribute may not be modified after an object is created.

The **CKA_CERTIFICATE_CATEGORY** and **CKA_TRUSTED** attributes will together be used to map to the categorization of the certificates.

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

The **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not attach any special meaning to them. When present, the application is responsible to set them to values that match the certificate's encoded "not before" and "not after" fields (if any).

4.6.3 X.509 public key certificate objects

X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 public key certificates. The following table defines the X.509 certificate object attributes, in addition to the common attributes defined for this object class:

Table 20, X.509 Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT ¹	Byte array	DER-encoding of the certificate subject name
CKA_ID	Byte array	Key identifier for public/private key pair (default empty)
CKA_ISSUER	Byte array	DER-encoding of the certificate issuer name (default empty)
CKA_SERIAL_NUMBER	Byte array	DER-encoding of the certificate serial number (default empty)
CKA_VALUE ²	Byte array	BER-encoding of the certificate
CKA_URL ³	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained (default empty)
CKA_HASH_OF_SUBJECT_PUBLIC_KEY ⁴	Byte array	Hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLIC_KEY ⁴	Byte array	Hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_JAVA_MIDP_SECURITY_DOMAIN	CK_JAVA_MIDP_SECURITY_DOMAIN	Java MIDP security domain. (default CK_SECURITY_DOMAIN_UNSPECIFIED)
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

¹MUST be specified when the object is created.

²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

³MUST be non-empty if CKA_VALUE is empty.

⁴Can only be empty if CKA_URL is empty.

Only the **CKA_ID**, **CKA_ISSUER**, and **CKA_SERIAL_NUMBER** attributes may be modified after the object is created.

The **CKA_ID** attribute is intended as a means of distinguishing multiple public-key/private-key pairs held by the same subject (whether stored in the same token or not). (Since the keys are distinguished by subject name as well as identifier, it is possible that keys for different subjects may have the same **CKA_ID** value without introducing any ambiguity.)

It is intended in the interests of interoperability that the subject name and key identifier for a certificate will be the same as those for the corresponding public and private keys (though it is not required that all be stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

The **CKA_ISSUER** and **CKA_SERIAL_NUMBER** attributes are for compatibility with PKCS #7 and Privacy Enhanced Mail (RFC1421). Note that with the version 3 extensions to X.509 certificates, the key identifier may be carried in the certificate. It is intended that the **CKA_ID** value be identical to the key identifier in such a certificate extension, although this will not be enforced by Cryptoki.

The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by **CKA_NAME_HASH_ALGORITHM**.

The **CKA_JAVA_MIDP_SECURITY_DOMAIN** attribute associates a certificate with a Java MIDP security domain.

The following is a sample template for creating an X.509 certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

4.6.4 WTLS public key certificate objects

WTLS certificate objects (certificate type **CKC_WTLS**) hold WTLS public key certificates. The following table defines the WTLS certificate object attributes, in addition to the common attributes defined for this object class.

Table 21: WTLS Certificate Object Attributes

Attribute	Data type	Meaning
CKA_SUBJECT ¹	Byte array	WTLS-encoding (Identifier type) of the certificate subject
CKA_ISSUER	Byte array	WTLS-encoding (Identifier type) of the certificate issuer (default empty)
CKA_VALUE ²	Byte array	WTLS-encoding of the certificate
CKA_URL ³	RFC2279 string	If not empty this attribute gives the URL where the complete certificate can be obtained
CKA_HASH_OF_SUBJECT_PUBLIC_KEY ⁴	Byte array	SHA-1 hash of the subject public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_HASH_OF_ISSUER_PUBLIC_KEY ⁴	Byte array	SHA-1 hash of the issuer public key (default empty). Hash algorithm is defined by CKA_NAME_HASH_ALGORITHM
CKA_NAME_HASH_ALGORITHM	CK_MECHANISM_TYPE	Defines the mechanism used to calculate CKA_HASH_OF_SUBJECT_PUBLIC

Attribute	Data type	Meaning
		_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

¹MUST be specified when the object is created. Can only be empty if CKA_VALUE is empty.

²MUST be specified when the object is created. MUST be non-empty if CKA_URL is empty.

³MUST be non-empty if CKA_VALUE is empty.

⁴Can only be empty if CKA_URL is empty.

Only the **CKA_ISSUER** attribute may be modified after the object has been created.

The encoding for the **CKA_SUBJECT**, **CKA_ISSUER**, and **CKA_VALUE** attributes can be found in [WTLS].

The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile environments.

The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY** attributes are used to store the hashes of the public keys of the subject and the issuer. They are particularly important when only the URL is available to be able to correlate a certificate with a private key and when searching for the certificate of the issuer. The hash algorithm is defined by CKA_NAME_HASH_ALGORITHM.

The following is a sample template for creating a WTLS certificate object:

```

CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_WTLS;
CK_UTF8CHAR label[] = "A certificate object";
CK_BYTE subject[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] =
{
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};

```

4.6.5 X.509 attribute certificate objects

X.509 attribute certificate objects (certificate type **CKC_X_509_ATTR_CERT**) hold X.509 attribute certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the common attributes defined for this object class:

1637 Table 22, X.509 Attribute Certificate Object Attributes

Attribute	Data Type	Meaning
CKA_OWNER ¹	Byte Array	DER-encoding of the attribute certificate's subject field. This is distinct from the CKA_SUBJECT attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different.
CKA_AC_ISSUER	Byte Array	DER-encoding of the attribute certificate's issuer field. This is distinct from the CKA_ISSUER attribute contained in CKC_X_509 certificates because the ASN.1 syntax and encoding are different. (default empty)
CKA_SERIAL_NUMBER	Byte Array	DER-encoding of the certificate serial number. (default empty)
CKA_ATTR_TYPES	Byte Array	BER-encoding of a sequence of object identifier values corresponding to the attribute types contained in the certificate. When present, this field offers an opportunity for applications to search for a particular attribute certificate without fetching and parsing the certificate itself. (default empty)
CKA_VALUE ¹	Byte Array	BER-encoding of the certificate.

1638 ¹MUST be specified when the object is created

1639 Only the **CKA_AC_ISSUER**, **CKA_SERIAL_NUMBER** and **CKA_ATTR_TYPES** attributes may be

1640 modified after the object is created.

1641 The following is a sample template for creating an X.509 attribute certificate object:

```
CK_OBJECT_CLASS class = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
CK_UTF8CHAR label[] = "An attribute certificate object";
CK_BYTE owner[] = {...};
CK_BYTE certificate[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OWNER, owner, sizeof(owner)},
    {CKA_VALUE, certificate, sizeof(certificate)}
};
```

1656 **4.7 Key objects**

1657 **4.7.1 Definitions**

1658 There is no CKO_ definition for the base key object class, only for the key types derived from it.

1659 This section defines the object class CKO_PUBLIC_KEY, CKO_PRIVATE_KEY and

1660 CKO_SECRET_KEY for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

1661 **4.7.2 Overview**

1662 Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret

1663 keys. The following common footnotes apply to all the tables describing attributes of keys:

1664 The following table defines the attributes common to public key, private key and secret key classes, in

1665 addition to the common attributes defined for this object class:

Attribute	Data Type	Meaning
CKA_KEY_TYPE ^{1,5}	CK_KEY_TYPE	Type of key
CKA_ID ⁸	Byte array	Key identifier for key (default empty)
CKA_START_DATE ⁸	CK_DATE	Start date for the key (default empty)
CKA_END_DATE ⁸	CK_DATE	End date for the key (default empty)
CKA_DERIVE ⁸	CK_BBOOL	CK_TRUE if key supports key derivation (<i>i.e.</i> , if other keys can be derived from this one (default CK_FALSE))
CKA_LOCAL ^{2,4,6}	CK_BBOOL	CK_TRUE only if key was either <ul style="list-style-type: none"> generated locally (<i>i.e.</i>, on the token) with a C_GenerateKey or C_GenerateKeyPair call created with a C_CopyObject call as a copy of a key which had its CKA_LOCAL attribute set to CK_TRUE
CKA_KEY_GEN_MECHANISM ^{2,4,6}	CK_MECHANISM_TYPE	Identifier of the mechanism used to generate the key material.
CKA_ALLOWED_MECHANISMS	CK_MECHANISM_TYPE_PTR, pointer to a CK_MECHANISM_TYPE array	A list of mechanisms allowed to be used with this key. The number of mechanisms in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_MECHANISM_TYPE.

1667 Refer to Table 11 for footnotes

1668 The **CKA_ID** field is intended to distinguish among multiple keys. In the case of public and private keys,
1669 this field assists in handling multiple keys held by the same subject; the key identifier for a public key and
1670 its corresponding private key should be the same. The key identifier should also be the same as for the
1671 corresponding certificate, if one exists. Cryptoki does not enforce these associations, however. (See
1672 Section 4.6 for further commentary.)

1673 In the case of secret keys, the meaning of the **CKA_ID** attribute is up to the application.

1674 Note that the **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does
1675 not attach any special meaning to them. In particular, it does not restrict usage of a key according to the
1676 dates; doing this is up to the application.

1677 The **CKA_DERIVE** attribute has the value CK_TRUE if and only if it is possible to derive other keys from
1678 the key.

1679 The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the value of the key was originally
1680 generated on the token by a **C_GenerateKey** or **C_GenerateKeyPair** call.

1681 The **CKA_KEY_GEN_MECHANISM** attribute identifies the key generation mechanism used to generate
1682 the key material. It contains a valid value only if the **CKA_LOCAL** attribute has the value CK_TRUE. If
1683 **CKA_LOCAL** has the value CK_FALSE, the value of the attribute is
1684 CK_UNAVAILABLE_INFORMATION.

1685 4.8 Public key objects

1686 Public key objects (object class **CKO_PUBLIC_KEY**) hold public keys. The following table defines the
1687 attributes common to all public keys, in addition to the common attributes defined for this object class:

Attribute	Data type	Meaning
CKA_SUBJECT ⁸	Byte array	DER-encoding of the key subject name (default empty)
CKA_ENCRYPT ⁸	CK_BBOOL	CK_TRUE if key supports encryption ⁹
CKA_VERIFY ⁸	CK_BBOOL	CK_TRUE if key supports verification where the signature is an appendix to the data ⁹
CKA_VERIFY_RECOVER ⁸	CK_BBOOL	CK_TRUE if key supports verification where the data is recovered from the signature ⁹
CKA_WRAP ⁸	CK_BBOOL	CK_TRUE if key supports wrapping (<i>i.e.</i> , can be used to wrap other keys) ⁹
CKA_TRUSTED ¹⁰	CK_BBOOL	The key can be trusted for the application that it was created. The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.
CKA_PUBLIC_KEY_INFO	Byte array	DER-encoding of the SubjectPublicKeyInfo for this public key. (MAY be empty, DEFAULT derived from the underlying public key data)

1689 Refer to Table 11 for footnotes

1690 It is intended in the interests of interoperability that the subject name and key identifier for a public key will
 1691 be the same as those for the corresponding certificate and private key. However, Cryptoki does not
 1692 enforce this, and it is not required that the certificate and private key also be stored on the token.

1693 To map between ISO/IEC 9594-8 (X.509) **keyUsage** flags for public keys and the PKCS #11 attributes for
 1694 public keys, use the following table.

1695 Table 25, Mapping of X.509 key usage flags to Cryptoki attributes for public keys

Key usage flags for public keys in X.509 public key certificates	Corresponding cryptoki attributes for public keys.
dataEncipherment	CKA_ENCRYPT
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY
digitalSignature, keyCertSign, cRLSign	CKA_VERIFY_RECOVER
keyAgreement	CKA_DERIVE
keyEncipherment	CKA_WRAP
nonRepudiation	CKA_VERIFY
nonRepudiation	CKA_VERIFY_RECOVER

1696 The value of the CKA_PUBLIC_KEY_INFO attribute is the DER encoded value of SubjectPublicKeyInfo:

1697 SubjectPublicKeyInfo ::= SEQUENCE {
1698 algorithm AlgorithmIdentifier,
1699 subjectPublicKey BIT_STRING }

1700 The encodings for the subjectPublicKey field are specified in the description of the public key types in the
1701 appropriate [PKCS11-Curr] document for the key types defined within this specification.

1702 4.9 Private key objects

1703 Private key objects (object class **CKO_PRIVATE_KEY**) hold private keys. The following table defines the
1704 attributes common to all private keys, in addition to the common attributes defined for this object class:

1705 Table 26, Common Private Key Attributes

Attribute	Data type	Meaning
CKA_SUBJECT ⁸	Byte array	DER-encoding of certificate subject name (default empty)
CKA_SENSITIVE ^{8,11}	CK_BBOOL	CK_TRUE if key is sensitive ⁹
CKA_DECRYPT ⁸	CK_BBOOL	CK_TRUE if key supports decryption ⁹
CKA_SIGN ⁸	CK_BBOOL	CK_TRUE if key supports signatures where the signature is an appendix to the data ⁹
CKA_SIGN_RECOVER ⁸	CK_BBOOL	CK_TRUE if key supports signatures where the data can be recovered from the signature ⁹
CKA_UNWRAP ⁸	CK_BBOOL	CK_TRUE if key supports unwrapping (<i>i.e.</i> , can be used to unwrap other keys) ⁹
CKA_EXTRACTABLE ^{8,12}	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped ⁹
CKA_ALWAYS_SENSITIVE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>never</i> had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_WRAP_WITH_TRUSTED ¹¹	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE.

Attribute	Data type	Meaning
		Default is CK_FALSE.
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.
CKA_ALWAYS_AUTHENTICATE	CK_BBOOL	If CK_TRUE, the user has to supply the PIN for each use (sign or decrypt) with the key. Default is CK_FALSE.
CKA_PUBLIC_KEY_INFO ⁸	Byte Array	DER-encoding of the SubjectPublicKeyInfo for the associated public key (MAY be empty; DEFAULT derived from the underlying private key data; MAY be manually set for specific key types; if set; MUST be consistent with the underlying private key data)
CKA_DERIVE_TEMPLATE	CK_ATTRIBUTE_PTR	For deriving keys. The attribute template to match against any keys derived using this derivation key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.

1706 Refer to Table 11 for footnotes

1707 It is intended in the interests of interoperability that the subject name and key identifier for a private key
1708 will be the same as those for the corresponding certificate and public key. However, this is not enforced
1709 by Cryptoki, and it is not required that the certificate and public key also be stored on the token.

1710 If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE,
1711 then certain attributes of the private key cannot be revealed in plaintext outside the token. Which
1712 attributes these are is specified for each type of private key in the attribute table in the section describing
1713 that type of key.

1714 The **CKA_ALWAYS_AUTHENTICATE** attribute can be used to force re-authentication (i.e. force the user
1715 to provide a PIN) for each use of a private key. "Use" in this case means a cryptographic operation such
1716 as sign or decrypt. This attribute may only be set to CK_TRUE when **CKA_PRIVATE** is also CK_TRUE.

1717 Re-authentication occurs by calling **C_Login** with *userType* set to **CKU_CONTEXT_SPECIFIC**
1718 immediately after a cryptographic operation using the key has been initiated (e.g. after **C_SignInit**). In
1719 this call, the actual user type is implicitly given by the usage requirements of the active key. If **C_Login**
1720 returns CKR_OK the user was successfully authenticated and this sets the active key in an authenticated
1721 state that lasts until the cryptographic operation has successfully or unsuccessfully been completed (e.g.

by **C_Sign**, **C_SignFinal**,...). A return value **CKR_PIN_INCORRECT** from **C_Login** means that the user was denied permission to use the key and continuing the cryptographic operation will result in a behavior as if **C_Login** had not been called. In both of these cases the session state will remain the same, however repeated failed re-authentication attempts may cause the PIN to be locked. **C_Login** returns in this case **CKR_PIN_LOCKED** and this also logs the user out from the token. Failing or omitting to re-authenticate when **CKA_ALWAYS_AUTHENTICATE** is set to **CK_TRUE** will result in **CKR_USER_NOT_LOGGED_IN** to be returned from calls using the key. **C_Login** will return **CKR_OPERATION_NOT_INITIALIZED**, but the active cryptographic operation will not be affected, if an attempt is made to re-authenticate when **CKA_ALWAYS_AUTHENTICATE** is set to **CK_FALSE**.

The **CKA_PUBLIC_KEY_INFO** attribute represents the public key associated with this private key. The data it represents may either be stored as part of the private key data, or regenerated as needed from the private key.

If this attribute is supplied as part of a template for **C_CreateObject**, **C_CopyObject** or **C_SetAttributeValue** for a private key, the token MUST verify correspondence between the private key data and the public key data as supplied in **CKA_PUBLIC_KEY_INFO**. This can be done either by deriving a public key from the private key and comparing the values, or by doing a sign and verify operation. If there is a mismatch, the command SHALL return **CKR_ATTRIBUTE_VALUE_INVALID**. A token MAY choose not to support the **CKA_PUBLIC_KEY_INFO** attribute for commands which create new private keys. If it does not support the attribute, the command SHALL return **CKR_ATTRIBUTE_TYPE_INVALID**.

As a general guideline, private keys of any type SHOULD store sufficient information to retrieve the public key information. In particular, the RSA private key description has been modified in PKCS #11 V2.40 to add the **CKA_PUBLIC_EXPONENT** to the list of attributes required for an RSA private key. All other private key types described in this specification contain sufficient information to recover the associated public key.

4.10 Secret key objects

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

Table 27, Common Secret Key Attributes

Attribute	Data type	Meaning
CKA_SENSITIVE ^{8,11}	CK_BBOOL	CK_TRUE if object is sensitive (default CK_FALSE)
CKA_ENCRYPT ⁸	CK_BBOOL	CK_TRUE if key supports encryption ⁹
CKA_DECRYPT ⁸	CK_BBOOL	CK_TRUE if key supports decryption ⁹
CKA_SIGN ⁸	CK_BBOOL	CK_TRUE if key supports signatures (<i>i.e.</i> , authentication codes) where the signature is an appendix to the data ⁹
CKA_VERIFY ⁸	CK_BBOOL	CK_TRUE if key supports verification (<i>i.e.</i> , of authentication codes) where the signature is an appendix to the data ⁹
CKA_WRAP ⁸	CK_BBOOL	CK_TRUE if key supports wrapping (<i>i.e.</i> , can be used to wrap other keys) ⁹
CKA_UNWRAP ⁸	CK_BBOOL	CK_TRUE if key supports unwrapping (<i>i.e.</i> , can be used to unwrap other keys) ⁹

Attribute	Data type	Meaning
CKA_EXTRACTABLE ^{8,12}	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped ⁹
CKA_ALWAYS_SENSITIVE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>always</i> had the CKA_SENSITIVE attribute set to CK_TRUE
CKA_NEVER_EXTRACTABLE ^{2,4,6}	CK_BBOOL	CK_TRUE if key has <i>never</i> had the CKA_EXTRACTABLE attribute set to CK_TRUE
CKA_CHECK_VALUE	Byte array	Key checksum
CKA_WRAP_WITH_TRUSTED ¹¹	CK_BBOOL	CK_TRUE if the key can only be wrapped with a wrapping key that has CKA_TRUSTED set to CK_TRUE. Default is CK_FALSE.
CKA_TRUSTED ¹⁰	CK_BBOOL	The wrapping key can be used to wrap keys with CKA_WRAP_WITH_TRUSTED set to CK_TRUE.
CKA_WRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to match against any keys wrapped using this wrapping key. Keys that do not match cannot be wrapped. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE
CKA_UNWRAP_TEMPLATE	CK_ATTRIBUTE_PTR	For wrapping keys. The attribute template to apply to any keys unwrapped using this wrapping key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.
A_DERIVE_TEMPLATE	CK_ATTRIBUTE_PTR	For deriving keys. The attribute template to match against any keys derived using this derivation key. Any user supplied template is applied after this template as if the object has already been created. The number of attributes in the array is the <i>ulValueLen</i> component of the attribute divided by the size of CK_ATTRIBUTE.

1751 Refer to Table 11 for footnotes

If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE, then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which attributes these are is specified for each type of secret key in the attribute table in the section describing that type of key.

The key check value (KCV) attribute for symmetric key objects to be called **CKA_CHECK_VALUE**, of type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to be used to cross-check symmetric keys against other systems where the same key is shared, and as a validity check after manual key entry or restore from backup. Refer to object definitions of specific key types for KCV algorithms.

Properties:

1. For two keys that are cryptographically identical the value of this attribute should be identical.
2. **CKA_CHECK_VALUE** should not be usable to obtain any part of the key value.
3. Non-uniqueness. Two different keys can have the same **CKA_CHECK_VALUE**. This is unlikely (the probability can easily be calculated) but possible.

The attribute is optional, but if supported, regardless of how the key object is created or derived, the value of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is forbidden (i.e. when **CKA_ENCRYPT** is set to CK_FALSE).

If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST match what the library calculates it to be or the library returns a **CKR_ATTRIBUTE_VALUE_INVALID**. If the library does not support the attribute then it should ignore it. Allowing the attribute in the template this way does no harm and allows the attribute to be treated like any other attribute for the purposes of key wrap and unwrap where the attributes are preserved also.

The generation of the KCV may be prevented by the application supplying the attribute in the template as a no-value (0 length) entry. The application can query the value at any time like any other attribute using **C_GetAttributeValue**. **C_SetAttributeValue** may be used to destroy the attribute, by supplying no-value.

Unless otherwise specified for the object definition, the value of this attribute is derived from the key object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

4.11 Domain parameter objects

4.11.1 Definitions

This section defines the object class **CKO_DOMAIN_PARAMETERS** for type **CK_OBJECT_CLASS** as used in the **CKA_CLASS** attribute of objects.

4.11.2 Overview

This object class was created to support the storage of certain algorithm's extended parameters. DSA and DH both use domain parameters in the key-pair generation step. In particular, some libraries support the generation of domain parameters (originally out of scope for PKCS11) so the object class was added.

To use a domain parameter object you MUST extract the attributes into a template and supply them (still in the template) to the corresponding key-pair generation function.

Domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**) hold public domain parameters.

The following table defines the attributes common to domain parameter objects in addition to the common attributes defined for this object class:

Table 28, Common Domain Parameter Attributes

Attribute	Data Type	Meaning
CKA_KEY_TYPE ¹	CK_KEY_TYPE	Type of key the domain parameters can be used to generate.
CKA_LOCAL ^{2,4}	CK_BBOOL	CK_TRUE only if domain parameters were either <ul style="list-style-type: none"> generated locally (<i>i.e.</i>, on the token) with a C_GenerateKey created with a C_CopyObject call as a copy of domain parameters which had its CKA_LOCAL attribute set to CK_TRUE

¹ Refer to Table 11 for footnotes

The **CKA_LOCAL** attribute has the value CK_TRUE if and only if the values of the domain parameters were originally generated on the token by a **C_GenerateKey** call.

4.12 Mechanism objects

4.12.1 Definitions

This section defines the object class CKO_MECHANISM for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.12.2 Overview

Mechanism objects provide information about mechanisms supported by a device beyond that given by the **CK_MECHANISM_INFO** structure.

When searching for objects using **C_FindObjectsInit** and **C_FindObjects**, mechanism objects are not returned unless the **CKA_CLASS** attribute in the template has the value **CKO_MECHANISM**. This protects applications written to previous versions of Cryptoki from finding objects that they do not understand.

Table 29, Common Mechanism Attributes

Attribute	Data Type	Meaning
CKA_MECHANISM_TYPE	CK_MECHANISM_TYPE	The type of mechanism object

The **CKA_MECHANISM_TYPE** attribute may not be set.

4.13 Profile objects

4.13.1 Definitions

This section defines the object class CKO_PROFILE for type CK_OBJECT_CLASS as used in the CKA_CLASS attribute of objects.

4.13.2 Overview

Profile objects (object class CKO_PROFILE) describe which PKCS #11 profiles the token implements. Profiles are defined in the OASIS PKCS #11 Cryptographic Token Interface Profiles document. A given token can contain more than one profile ID. The following table lists the attributes supported by profile objects, in addition to the common attributes defined for this object class:

Table 30, Profile Object Attributes

Attribute	Data type	Meaning
CKA_PROFILE_ID	CK_PROFILE_ID	ID of the supported profile.

1821 The **CKA_PROFILE_ID** attribute identifies a profile that the token supports.

5 Functions

Cryptoki's functions are organized into the following categories:

- general-purpose functions (4 functions)
- slot and token management functions (9 functions)
- session management functions (8 functions)
- object management functions (9 functions)
- encryption functions (4 functions)
- message-based encryption functions (5 functions)
- decryption functions (4 functions)
- message digesting functions (5 functions)
- signing and MACing functions (6 functions)
- functions for verifying signatures and MACs (6 functions)
- dual-purpose cryptographic functions (4 functions)
- key management functions (5 functions)
- random number generation functions (2 functions)
- parallel function management functions (2 functions)

In addition to these functions, Cryptoki can use application-supplied callback functions to notify an application of certain events, and can also use application-supplied functions to handle mutex objects for safe multi-threaded library access.

The Cryptoki API functions are presented in the following table:

Table 31, Summary of Cryptoki Functions

Category	Function	Description
General purpose functions	C_Initialize	initializes Cryptoki
	C_Finalize	clean up miscellaneous Cryptoki-associated resources
	C_GetInfo	obtains general information about Cryptoki
	C_GetFunctionList	obtains entry points of Cryptoki library functions
	C_GetInterfaceList	obtains list of interfaces supported by Cryptoki library
	C_GetInterface	obtains interface specific entry points to Cryptoki library functions
Slot and token management functions	C_GetSlotList	obtains a list of slots in the system
	C_GetSlotInfo	obtains information about a particular slot
	C_GetTokenInfo	obtains information about a particular token
	C_WaitForSlotEvent	waits for a slot event (token insertion, removal, etc.) to occur
	C_GetMechanismList	obtains a list of mechanisms supported by a token
	C_GetMechanismInfo	obtains information about a particular mechanism
	C_InitToken	initializes a token
	C_InitPIN	initializes the normal user's PIN

Category	Function	Description
	C_SetPIN	modifies the PIN of the current user
Session management functions	C_OpenSession	opens a connection between an application and a particular token or sets up an application callback for token insertion
	C_CloseSession	closes a session
	C_CloseAllSessions	closes all sessions with a token
	C_GetSessionInfo	obtains information about the session
	C_SessionCancel	terminates active session based operations
	C_GetOperationState	obtains the cryptographic operations state of a session
	C_SetOperationState	sets the cryptographic operations state of a session
	C_Login	logs into a token
	C_LoginUser	logs into a token with explicit user name
	C_Logout	logs out from a token
Object management functions	C_CreateObject	creates an object
	C_CopyObject	creates a copy of an object
	C_DestroyObject	destroys an object
	C_GetObjectSize	obtains the size of an object in bytes
	C_GetAttributeValue	obtains an attribute value of an object
	C_SetAttributeValue	modifies an attribute value of an object
	C_FindObjectsInit	initializes an object search operation
	C_FindObjects	continues an object search operation
	C_FindObjectsFinal	finishes an object search operation
Encryption functions	C_EncryptInit	initializes an encryption operation
	C_Encrypt	encrypts single-part data
	C_EncryptUpdate	continues a multiple-part encryption operation
	C_EncryptFinal	finishes a multiple-part encryption operation
Message-based Encryption Functions	C_MessageEncryptInit	initializes a message-based encryption process
	C_EncryptMessage	encrypts a single-part message
	C_EncryptMessageBegin	begins a multiple-part message encryption operation
	C_EncryptMessageNext	continues or finishes a multiple-part message encryption operation
	C_MessageEncryptFinal	finishes a message-based encryption process
Decryption Functions	C_DecryptInit	initializes a decryption operation
	C_Decrypt	decrypts single-part encrypted data
	C_DecryptUpdate	continues a multiple-part decryption operation
	C_DecryptFinal	finishes a multiple-part decryption operation
Message-based Decryption Functions	C_MessageDecryptInit	initializes a message decryption operation
	C_DecryptMessage	decrypts single-part data
	C_DecryptMessageBegin	starts a multiple-part message decryption operation
	C_DecryptMessageNext	Continues and finishes a multiple-part message decryption operation

Category	Function	Description
	C_MessageDecryptFinal	finishes a message decryption operation
Message Digesting Functions	C_DigestInit	initializes a message-digesting operation
	C_Digest	digests single-part data
	C_DigestUpdate	continues a multiple-part digesting operation
	C_DigestKey	digests a key
	C_DigestFinal	finishes a multiple-part digesting operation
Signing and MACing functions	C_SignInit	initializes a signature operation
	C_Sign	signs single-part data
	C_SignUpdate	continues a multiple-part signature operation
	C_SignFinal	finishes a multiple-part signature operation
	C_SignRecoverInit	initializes a signature operation, where the data can be recovered from the signature
	C_SignRecover	signs single-part data, where the data can be recovered from the signature
Message-based Signature functions	C_MessageSignInit	initializes a message signature operation
	C_SignMessage	signs single-part data
	C_SignMessageBegin	starts a multiple-part message signature operation
	C_SignMessageNext	continues and finishes a multiple-part message signature operation
	C_MessageSignFinal	finishes a message signature operation
Functions for verifying signatures and MACs	C_VerifyInit	initializes a verification operation
	C_Verify	verifies a signature on single-part data
	C_VerifyUpdate	continues a multiple-part verification operation
	C_VerifyFinal	finishes a multiple-part verification operation
	C_VerifyRecoverInit	initializes a verification operation where the data is recovered from the signature
	C_VerifyRecover	verifies a signature on single-part data, where the data is recovered from the signature
Message-based Functions for verifying signatures and MACs	C_MessageVerifyInit	initializes a message verification operation
	C_VerifyMessage	verifies single-part data
	C_VerifyMessageBegin	starts a multiple-part message verification operation
	C_VerifyMessageNext	continues and finishes a multiple-part message verification operation
	C_MessageVerifyFinal	finishes a message verification operation
Dual-purpose cryptographic functions	C_DigestEncryptUpdate	continues simultaneous multiple-part digesting and encryption operations
	C_DecryptDigestUpdate	continues simultaneous multiple-part decryption and digesting operations
	C_SignEncryptUpdate	continues simultaneous multiple-part signature and encryption operations
	C_DecryptVerifyUpdate	continues simultaneous multiple-part decryption and verification operations
Key management	C_GenerateKey	generates a secret key
	C_GenerateKeyPair	generates a public-key/private-key pair

Category	Function	Description
functions	C_WrapKey	wraps (encrypts) a key
	C_UnwrapKey	unwraps (decrypts) a key
	C_DeriveKey	derives a key from a base key
Random number generation functions	C_SeedRandom	mixes in additional seed material to the random number generator
	C_GenerateRandom	generates random data
Parallel function management functions	C_GetFunctionStatus	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
	C_CancelFunction	legacy function which always returns CKR_FUNCTION_NOT_PARALLEL
Callback function		application-supplied function to process notifications from Cryptoki

Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes either its entire goal, or nothing at all.

- If a Cryptoki function executes successfully, it returns the value CKR_OK.
- If a Cryptoki function does not execute successfully, it returns some value other than CKR_OK, and the token is in the same state as it was in prior to the function call. If the function call was supposed to modify the contents of certain memory addresses on the host computer, these memory addresses may have been modified, despite the failure of the function.
- In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value CKR_GENERAL_ERROR. When this happens, the token and/or host computer may be in an inconsistent state, and the goals of the function may have been partially achieved.

There are a small number of Cryptoki functions whose return values do not behave precisely as described above; these exceptions are documented individually with the description of the functions themselves.

A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported function **MUST** have a “stub” in the library which simply returns the value CKR_FUNCTION_NOT_SUPPORTED. The function’s entry in the library’s **CK_FUNCTION_LIST** structure (as obtained by **C_GetFunctionList**) should point to this stub function (see Section 3.6).

5.1 Function return values

The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1 details the behavior of Cryptoki functions, including what values each of them may return.

Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications attempt to give some leeway when interpreting Cryptoki functions’ return values. We have attempted to specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are presumably some gaps. For example, it is possible that a particular error code which might apply to a particular Cryptoki function is unfortunately not actually listed in the description of that function as a possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that error code for that function. It would be far preferable for the application to examine the function’s return value, see that it indicates some sort of error (even if the application doesn’t know precisely *what* kind of error), and behave accordingly.

See Section 5.1.8 for some specific details on how a developer might attempt to make an application that accommodates a range of behaviors from Cryptoki libraries.

5.1.1 Universal Cryptoki function return values

Any Cryptoki function can return any of the following values:

- **CKR_GENERAL_ERROR**: Some horrible, unrecoverable error has occurred. In the worst case, it is possible that the function only partially succeeded, and that the computer and/or token is in an inconsistent state.
- **CKR_HOST_MEMORY**: The computer that the Cryptoki library is running on has insufficient memory to perform the requested function.
- **CKR_FUNCTION_FAILED**: The requested function could not be performed, but detailed information about why not is not available in this error return. If the failed function uses a session, it is possible that the **CK_SESSION_INFO** structure that can be obtained by calling **C_GetSessionInfo** will hold useful information about what happened in its *ulDeviceError* field. In any event, although the function call failed, the situation is not necessarily totally hopeless, as it is likely to be when **CKR_GENERAL_ERROR** is returned. Depending on what the root cause of the error actually was, it is possible that an attempt to make the exact same function call again would succeed.
- **CKR_OK**: The function executed successfully. Technically, **CKR_OK** is not *quite* a “universal” return value; in particular, the legacy functions **C_GetFunctionStatus** and **C_CancelFunction** (see Section 5.20) cannot return **CKR_OK**.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of **CKR_GENERAL_ERROR** or **CKR_HOST_MEMORY** would be an appropriate error return, then **CKR_GENERAL_ERROR** should be returned.

5.1.2 Cryptoki function return values for functions that use a session handle

Any Cryptoki function that takes a session handle as one of its arguments (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, **C_GetTokenInfo**, **C_WaitForSlotEvent**, **C_GetMechanismList**, **C_GetMechanismInfo**, **C_InitToken**, **C_OpenSession**, and **C_CloseAllSessions**) can return the following values:

- **CKR_SESSION_HANDLE_INVALID**: The specified session handle was invalid *at the time that the function was invoked*. Note that this can happen if the session's token is removed before the function invocation, since removing a token closes all sessions with it.
- **CKR_DEVICE_REMOVED**: The token was removed from its slot *during the execution of the function*.
- **CKR_SESSION_CLOSED**: The session was closed *during the execution of the function*. Note that, as stated in **[PKCS11-UG]**, the behavior of Cryptoki is *undefined* if multiple threads of an application attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no guarantee that a function invocation could ever return the value **CKR_SESSION_CLOSED**. An example of multiple threads accessing a common session simultaneously is where one thread is using a session when another thread closes that same session.

The relative priorities of these errors are in the order listed above, *e.g.*, if either of **CKR_SESSION_HANDLE_INVALID** or **CKR_DEVICE_REMOVED** would be an appropriate error return, then **CKR_SESSION_HANDLE_INVALID** should be returned.

In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function execution.

5.1.3 Cryptoki function return values for functions that use a token

Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, or **C_WaitForSlotEvent**) can return any of the following values:

- **CKR_DEVICE_MEMORY**: The token does not have sufficient memory to perform the requested function.

- 1927 • CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot. This error code can
1928 be returned by more than just the functions mentioned above; in particular, it is possible for
1929 **C_GetSlotInfo** to return CKR_DEVICE_ERROR.
- 1930 • CKR_TOKEN_NOT_PRESENT: The token was not present in its slot *at the time that the function was*
1931 *invoked*.
- 1932 • CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.
- 1933 The relative priorities of these errors are in the order listed above, e.g., if either of
1934 CKR_DEVICE_MEMORY or CKR_DEVICE_ERROR would be an appropriate error return, then
1935 CKR_DEVICE_MEMORY should be returned.
- 1936 In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction
1937 between a token being removed *before* a function invocation and a token being removed *during* a
1938 function execution.

1939 5.1.4 Special return value for application-supplied callbacks

- 1940 There is a special-purpose return value which is not returned by any function in the actual Cryptoki API,
1941 but which may be returned by an application-supplied callback function. It is:
- 1942 • CKR_CANCEL: When a function executing in serial with an application decides to give the application
1943 a chance to do some work, it calls an application-supplied function with a CKN_SURRENDER
1944 callback (see Section 5.21). If the callback returns the value CKR_CANCEL, then the function aborts
1945 and returns CKR_FUNCTION_CANCELED.

1946 5.1.5 Special return values for mutex-handling functions

- 1947 There are two other special-purpose return values which are not returned by any actual Cryptoki
1948 functions. These values may be returned by application-supplied mutex-handling functions, and they may
1949 safely be ignored by application developers who are not using their own threading model. They are:
- 1950 • CKR_MUTEX_BAD: This error code can be returned by mutex-handling functions that are passed a
1951 bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a
1952 bad mutex object. There is therefore no guarantee that such a function will successfully detect bad
1953 mutex objects and return this value.
 - 1954 • CKR_MUTEX_NOT_LOCKED: This error code can be returned by mutex-unlocking functions. It
1955 indicates that the mutex supplied to the mutex-unlocking function was not locked.

1956 5.1.6 All other Cryptoki function return values

- 1957 Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions
1958 of particular error codes, there are in general no particular priorities among the errors listed below, i.e., if
1959 more than one error code might apply to an execution of a function, then the function may return any
1960 applicable error code.
- 1961 • CKR_ACTION_PROHIBITED: This value can only be returned by C_CopyObject,
1962 C_SetAttributeValue and C_DestroyObject. It denotes that the action may not be taken, either
1963 because of underlying policy restrictions on the token, or because the object has the relevant
1964 CKA_COPYABLE, CKA_MODIFIABLE or CKA_DESTROYABLE policy attribute set to CK_FALSE.
 - 1965 • CKR_ARGUMENTS_BAD: This is a rather generic error code which indicates that the arguments
1966 supplied to the Cryptoki function were in some way not appropriate.
 - 1967 • CKR_ATTRIBUTE_READ_ONLY: An attempt was made to set a value for an attribute which may not
1968 be set by the application, or which may not be modified by the application. See Section 4.1 for more
1969 information.
 - 1970 • CKR_ATTRIBUTE_SENSITIVE: An attempt was made to obtain the value of an attribute of an object
1971 which cannot be satisfied because the object is either sensitive or un-extractable.

- 1972 • CKR_ATTRIBUTE_TYPE_INVALID: An invalid attribute type was specified in a template. See
1973 Section 4.1 for more information.
- 1974 • CKR_ATTRIBUTE_VALUE_INVALID: An invalid value was specified for a particular attribute in a
1975 template. See Section 4.1 for more information.
- 1976 • CKR_BUFFER_TOO_SMALL: The output of the function is too large to fit in the supplied buffer.
- 1977 • CKR_CANT_LOCK: This value can only be returned by **C_Initialize**. It means that the type of locking
1978 requested by the application for thread-safety is not available in this library, and so the application
1979 cannot make use of this library in the specified fashion.
- 1980 • CKR_CRYPTOKI_ALREADY_INITIALIZED: This value can only be returned by **C_Initialize**. It
1981 means that the Cryptoki library has already been initialized (by a previous call to **C_Initialize** which
1982 did not have a matching **C_Finalize** call).
- 1983 • CKR_CRYPTOKI_NOT_INITIALIZED: This value can be returned by any function other than
1984 **C_Initialize**, **C_GetFunctionList**, **C_GetInterfaceList** and **C_GetInterface**. It indicates that the
1985 function cannot be executed because the Cryptoki library has not yet been initialized by a call to
1986 **C_Initialize**.
- 1987 • CKR_CURVE_NOT_SUPPORTED: This curve is not supported by this token. Used with Elliptic
1988 Curve mechanisms.
- 1989 • CKR_DATA_INVALID: The plaintext input data to a cryptographic operation is invalid. This return
1990 value has lower priority than CKR_DATA_LEN_RANGE.
- 1991 • CKR_DATA_LEN_RANGE: The plaintext input data to a cryptographic operation has a bad length.
1992 Depending on the operation's mechanism, this could mean that the plaintext data is too short, too
1993 long, or is not a multiple of some particular block size. This return value has higher priority than
1994 CKR_DATA_INVALID.
- 1995 • CKR_DOMAIN_PARAMS_INVALID: Invalid or unsupported domain parameters were supplied to the
1996 function. Which representation methods of domain parameters are supported by a given mechanism
1997 can vary from token to token.
- 1998 • CKR_ENCRYPTED_DATA_INVALID: The encrypted input to a decryption operation has been
1999 determined to be invalid ciphertext. This return value has lower priority than
2000 CKR_ENCRYPTED_DATA_LEN_RANGE.
- 2001 • CKR_ENCRYPTED_DATA_LEN_RANGE: The ciphertext input to a decryption operation has been
2002 determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's
2003 mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some
2004 particular block size. This return value has higher priority than CKR_ENCRYPTED_DATA_INVALID.
- 2005 • CKR_EXCEEDED_MAX_ITERATIONS: An iterative algorithm (for key pair generation, domain
2006 parameter generation etc.) failed because we have exceeded the maximum number of iterations.
2007 This error code has precedence over CKR_FUNCTION_FAILED. Examples of iterative algorithms
2008 include DSA signature generation (retry if either $r = 0$ or $s = 0$) and generation of DSA primes p and q
2009 specified in FIPS 186-4.
- 2010 • CKR_FIPS_SELF_TEST_FAILED: A FIPS 140-2 power-up self-test or conditional self-test failed.
2011 The token entered an error state. Future calls to cryptographic functions on the token will return
2012 CKR_GENERAL_ERROR. CKR_FIPS_SELF_TEST_FAILED has a higher precedence over
2013 CKR_GENERAL_ERROR. This error may be returned by **C_Initialize**, if a power-up self-test failed,
2014 by **C_GenerateRandom** or **C_SeedRandom**, if the continuous random number generator test failed,
2015 or by **C_GenerateKeyPair**, if the pair-wise consistency test failed.
- 2016 • CKR_FUNCTION_CANCELED: The function was canceled in mid-execution. This happens to a
2017 cryptographic function if the function makes a **CKN_SURRENDER** application callback which returns
2018 CKR_CANCEL (see CKR_CANCEL). It also happens to a function that performs PIN entry through a
2019 protected path. The method used to cancel a protected path PIN entry operation is device dependent.
- 2020 • CKR_FUNCTION_NOT_PARALLEL: There is currently no function executing in parallel in the
2021 specified session. This is a legacy error code which is only returned by the legacy functions
2022 **C_GetFunctionStatus** and **C_CancelFunction**.

- 2023 • **CKR_FUNCTION_NOT_SUPPORTED**: The requested function is not supported by this Cryptoki
2024 library. Even unsupported functions in the Cryptoki API should have a “stub” in the library; this stub
2025 should simply return the value **CKR_FUNCTION_NOT_SUPPORTED**.
- 2026 • **CKR_FUNCTION_REJECTED**: The signature request is rejected by the user.
- 2027 • **CKR_INFORMATION_SENSITIVE**: The information requested could not be obtained because the
2028 token considers it sensitive, and is not able or willing to reveal it.
- 2029 • **CKR_KEY_CHANGED**: This value is only returned by **C_SetOperationState**. It indicates that one of
2030 the keys specified is not the same key that was being used in the original saved session.
- 2031 • **CKR_KEY_FUNCTION_NOT_PERMITTED**: An attempt has been made to use a key for a
2032 cryptographic purpose that the key’s attributes are not set to allow it to do. For example, to use a key
2033 for performing encryption, that key **MUST** have its **CKA_ENCRYPT** attribute set to **CK_TRUE** (the
2034 fact that the key **MUST** have a **CKA_ENCRYPT** attribute implies that the key cannot be a private
2035 key). This return value has lower priority than **CKR_KEY_TYPE_INCONSISTENT**.
- 2036 • **CKR_KEY_HANDLE_INVALID**: The specified key handle is not valid. It may be the case that the
2037 specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a
2038 valid key handle.
- 2039 • **CKR_KEY_INDIGESTIBLE**: This error code can only be returned by **C_DigestKey**. It indicates that
2040 the value of the specified key cannot be digested for some reason (perhaps the key isn’t a secret key,
2041 or perhaps the token simply can’t digest this kind of key).
- 2042 • **CKR_KEY_NEEDED**: This value is only returned by **C_SetOperationState**. It indicates that the
2043 session state cannot be restored because **C_SetOperationState** needs to be supplied with one or
2044 more keys that were being used in the original saved session.
- 2045 • **CKR_KEY_NOT_NEEDED**: An extraneous key was supplied to **C_SetOperationState**. For
2046 example, an attempt was made to restore a session that had been performing a message digesting
2047 operation, and an encryption key was supplied.
- 2048 • **CKR_KEY_NOT_WRAPPABLE**: Although the specified private or secret key does not have its
2049 **CKA_EXTRACTABLE** attribute set to **CK_FALSE**, Cryptoki (or the token) is unable to wrap the key as
2050 requested (possibly the token can only wrap a given key with certain types of keys, and the wrapping
2051 key specified is not one of these types). Compare with **CKR_KEY_UNEXTRACTABLE**.
- 2052 • **CKR_KEY_SIZE_RANGE**: Although the requested keyed cryptographic operation could in principle
2053 be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key’s
2054 size is outside the range of key sizes that it can handle.
- 2055 • **CKR_KEY_TYPE_INCONSISTENT**: The specified key is not the correct type of key to use with the
2056 specified mechanism. This return value has a higher priority than
2057 **CKR_KEY_FUNCTION_NOT_PERMITTED**.
- 2058 • **CKR_KEY_UNEXTRACTABLE**: The specified private or secret key can’t be wrapped because its
2059 **CKA_EXTRACTABLE** attribute is set to **CK_FALSE**. Compare with **CKR_KEY_NOT_WRAPPABLE**.
- 2060 • **CKR_LIBRARY_LOAD_FAILED**: The Cryptoki library could not load a dependent shared library.
- 2061 • **CKR_MECHANISM_INVALID**: An invalid mechanism was specified to the cryptographic operation.
2062 This error code is an appropriate return value if an unknown mechanism was specified or if the
2063 mechanism specified cannot be used in the selected token with the selected function.
- 2064 • **CKR_MECHANISM_PARAM_INVALID**: Invalid parameters were supplied to the mechanism specified
2065 to the cryptographic operation. Which parameter values are supported by a given mechanism can
2066 vary from token to token.
- 2067 • **CKR_NEED_TO_CREATE_THREADS**: This value can only be returned by **C_Initialize**. It is
2068 returned when two conditions hold:
 - 2069 1. The application called **C_Initialize** in a way which tells the Cryptoki library that application
2070 threads executing calls to the library cannot use native operating system methods to spawn new
2071 threads.

- 2072 2. The library cannot function properly without being able to spawn new threads in the above
2073 fashion.
- 2074 • **CKR_NO_EVENT**: This value can only be returned by **C_WaitForSlotEvent**. It is returned when
2075 **C_WaitForSlotEvent** is called in non-blocking mode and there are no new slot events to return.
 - 2076 • **CKR_OBJECT_HANDLE_INVALID**: The specified object handle is not valid. We reiterate here that 0
2077 is never a valid object handle.
 - 2078 • **CKR_OPERATION_ACTIVE**: There is already an active operation (or combination of active
2079 operations) which prevents Cryptoki from activating the specified operation. For example, an active
2080 object-searching operation would prevent Cryptoki from activating an encryption operation with
2081 **C_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent
2082 Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous
2083 dual cryptographic operations in a session (see the description of the
2084 **CKF_DUAL_CRYPTO_OPERATIONS** flag in the **CK_TOKEN_INFO** structure), an active signature
2085 operation would prevent Cryptoki from activating an encryption operation.
 - 2086 • **CKR_OPERATION_NOT_INITIALIZED**: There is no active operation of an appropriate type in the
2087 specified session. For example, an application cannot call **C_Encrypt** in a session without having
2088 called **C_EncryptInit** first to activate an encryption operation.
 - 2089 • **CKR_PIN_EXPIRED**: The specified PIN has expired, and the requested operation cannot be carried
2090 out unless **C_SetPIN** is called to change the PIN value. Whether or not the normal user's PIN on a
2091 token ever expires varies from token to token.
 - 2092 • **CKR_PIN_INCORRECT**: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the
2093 token. More generally-- when authentication to the token involves something other than a PIN-- the
2094 attempt to authenticate the user has failed.
 - 2095 • **CKR_PIN_INVALID**: The specified PIN has invalid characters in it. This return code only applies to
2096 functions which attempt to set a PIN.
 - 2097 • **CKR_PIN_LEN_RANGE**: The specified PIN is too long or too short. This return code only applies to
2098 functions which attempt to set a PIN.
 - 2099 • **CKR_PIN_LOCKED**: The specified PIN is "locked", and cannot be used. That is, because some
2100 particular number of failed authentication attempts has been reached, the token is unwilling to permit
2101 further attempts at authentication. Depending on the token, the specified PIN may or may not remain
2102 locked indefinitely.
 - 2103 • **CKR_PIN_TOO_WEAK**: The specified PIN is too weak so that it could be easy to guess. If the PIN is
2104 too short, **CKR_PIN_LEN_RANGE** should be returned instead. This return code only applies to
2105 functions which attempt to set a PIN.
 - 2106 • **CKR_PUBLIC_KEY_INVALID**: The public key fails a public key validation. For example, an EC
2107 public key fails the public key validation specified in Section 5.2.2 of ANSI X9.62. This error code may
2108 be returned by **C_CreateObject**, when the public key is created, or by **C_VerifyInit** or
2109 **C_VerifyRecoverInit**, when the public key is used. It may also be returned by **C_DeriveKey**, in
2110 preference to **CKR_MECHANISM_PARAM_INVALID**, if the other party's public key specified in the
2111 mechanism's parameters is invalid.
 - 2112 • **CKR_RANDOM_NO_RNG**: This value can be returned by **C_SeedRandom** and
2113 **C_GenerateRandom**. It indicates that the specified token doesn't have a random number generator.
2114 This return value has higher priority than **CKR_RANDOM_SEED_NOT_SUPPORTED**.
 - 2115 • **CKR_RANDOM_SEED_NOT_SUPPORTED**: This value can only be returned by **C_SeedRandom**.
2116 It indicates that the token's random number generator does not accept seeding from an application.
2117 This return value has lower priority than **CKR_RANDOM_NO_RNG**.
 - 2118 • **CKR_SAVED_STATE_INVALID**: This value can only be returned by **C_SetOperationState**. It
2119 indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be
2120 restored to the specified session.

- 2121 • CKR_SESSION_COUNT: This value can only be returned by **C_OpenSession**. It indicates that the
2122 attempt to open a session failed, either because the token has too many sessions already open, or
2123 because the token has too many read/write sessions already open.
- 2124 • CKR_SESSION_EXISTS: This value can only be returned by **C_InitToken**. It indicates that a
2125 session with the token is already open, and so the token cannot be initialized.
- 2126 • CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel
2127 sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel
2128 sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by
2129 **C_OpenSession**, and it is only returned when **C_OpenSession** is called in a particular [deprecated]
2130 way.
- 2131 • CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action
2132 because it is a read-only session. This return value has lower priority than
2133 CKR_TOKEN_WRITE_PROTECTED.
- 2134 • CKR_SESSION_READ_ONLY_EXISTS: A read-only session already exists, and so the SO cannot
2135 be logged in.
- 2136 • CKR_SESSION_READ_WRITE_SO_EXISTS: A read/write SO session already exists, and so a
2137 read-only session cannot be opened.
- 2138 • CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid solely on
2139 the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.
- 2140 • CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower
2141 priority than CKR_SIGNATURE_LEN_RANGE.
- 2142 • CKR_SLOT_ID_INVALID: The specified slot ID is not valid.
- 2143 • CKR_STATE_UNSAVEABLE: The cryptographic operations state of the specified session cannot be
2144 saved for some reason (possibly the token is simply unable to save the current state). This return
2145 value has lower priority than CKR_OPERATION_NOT_INITIALIZED.
- 2146 • CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and
2147 lacks some necessary attributes. See Section 4.1 for more information.
- 2148 • CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting
2149 attributes. See Section 4.1 for more information.
- 2150 • CKR_TOKEN_NOT_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in
2151 the slot.
- 2152 • CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the
2153 token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.
- 2154 • CKR_UNWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_UnwrapKey**.
2155 It indicates that the key handle specified to be used to unwrap another key is not valid.
- 2156 • CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_UnwrapKey**. It
2157 indicates that although the requested unwrapping operation could in principle be carried out, this
2158 Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the
2159 range of key sizes that it can handle.
- 2160 • CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by
2161 **C_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent
2162 with the mechanism specified for unwrapping.
- 2163 • CKR_USER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that
2164 the specified user cannot be logged into the session, because it is already logged into the session.
2165 For example, if an application has an open SO session, and it attempts to log the SO into it, it will
2166 receive this error code.
- 2167 • CKR_USER_ANOTHER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It
2168 indicates that the specified user cannot be logged into the session, because another user is already

2169 logged into the session. For example, if an application has an open SO session, and it attempts to
 2170 log the normal user into it, it will receive this error code.

- 2171 • **CKR_USER_NOT_LOGGED_IN**: The desired action cannot be performed because the appropriate
 2172 user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out
 2173 unless it is logged in. Another example is that a private object cannot be created on a token unless
 2174 the session attempting to create it is logged in as the normal user. A final example is that
 2175 cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- 2176 • **CKR_USER_PIN_NOT_INITIALIZED**: This value can only be returned by **C_Login**. It indicates that
 2177 the normal user's PIN has not yet been initialized with **C_InitPIN**.
- 2178 • **CKR_USER_TOO_MANY_TYPES**: An attempt was made to have more distinct users simultaneously
 2179 logged into the token than the token and/or library permits. For example, if some application has an
 2180 open SO session, and another application attempts to log the normal user into a session, the attempt
 2181 may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be
 2182 supported does **C_Login** have to return this value. Note that this error code generalizes to true multi-
 2183 user tokens.
- 2184 • **CKR_USER_TYPE_INVALID**: An invalid value was specified as a **CK_USER_TYPE**. Valid types are
 2185 **CKU_SO**, **CKU_USER**, and **CKU_CONTEXT_SPECIFIC**.
- 2186 • **CKR_WRAPPED_KEY_INVALID**: This value can only be returned by **C_UnwrapKey**. It indicates
 2187 that the provided wrapped key is not valid. If a call is made to **C_UnwrapKey** to unwrap a particular
 2188 type of key (*i.e.*, some particular key type is specified in the template provided to **C_UnwrapKey**),
 2189 and the wrapped key provided to **C_UnwrapKey** is recognizably not a wrapped key of the proper
 2190 type, then **C_UnwrapKey** should return **CKR_WRAPPED_KEY_INVALID**. This return value has
 2191 lower priority than **CKR_WRAPPED_KEY_LEN_RANGE**.
- 2192 • **CKR_WRAPPED_KEY_LEN_RANGE**: This value can only be returned by **C_UnwrapKey**. It
 2193 indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.
 2194 This return value has higher priority than **CKR_WRAPPED_KEY_INVALID**.
- 2195 • **CKR_WRAPPING_KEY_HANDLE_INVALID**: This value can only be returned by **C_WrapKey**. It
 2196 indicates that the key handle specified to be used to wrap another key is not valid.
- 2197 • **CKR_WRAPPING_KEY_SIZE_RANGE**: This value can only be returned by **C_WrapKey**. It indicates
 2198 that although the requested wrapping operation could in principle be carried out, this Cryptoki library
 2199 (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range
 2200 of key sizes that it can handle.
- 2201 • **CKR_WRAPPING_KEY_TYPE_INCONSISTENT**: This value can only be returned by **C_WrapKey**. It
 2202 indicates that the type of the key specified to wrap another key is not consistent with the mechanism
 2203 specified for wrapping.
- 2204 • **CKR_OPERATION_CANCEL_FAILED**: This value can only be returned by **C_SessionCancel**. It
 2205 means that one or more of the requested operations could not be cancelled for implementation or
 2206 vendor-specific reasons.

2207 5.1.7 More on relative priorities of Cryptoki errors

2208 In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than **CKR_OK**) take
 2209 precedence over error codes from Section 5.1.2, which take precedence over error codes from Section
 2210 5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that
 2211 functions that use a session handle (*i.e.*, *most* functions!) never return the error code
 2212 **CKR_TOKEN_NOT_PRESENT** (they return **CKR_SESSION_HANDLE_INVALID** instead). Other than
 2213 these precedences, if more than one error code applies to the result of a Cryptoki call, any of the
 2214 applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the
 2215 descriptions of functions.

5.1.8 Error code “gotchas”

Here is a short list of a few particular things about return values that Cryptoki developers might want to be aware of:

1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction between a token being removed *before* a function invocation and a token being removed *during* a function invocation.
2. As mentioned in Section 5.1.2, an application should never count on getting a CKR_SESSION_CLOSED error.
3. The difference between CKR_DATA_INVALID and CKR_DATA_LEN_RANGE can be somewhat subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to always treat them equivalently.
4. Similarly, the difference between CKR_ENCRYPTED_DATA_INVALID and CKR_ENCRYPTED_DATA_LEN_RANGE, and between CKR_WRAPPED_KEY_INVALID and CKR_WRAPPED_KEY_LEN_RANGE, can be subtle, and it may be best to treat these return values equivalently.
5. Even with the guidance of Section 4.1, it can be difficult for a Cryptoki library developer to know which of CKR_ATTRIBUTE_VALUE_INVALID, CKR_TEMPLATE_INCOMPLETE, or CKR_TEMPLATE_INCONSISTENT to return. When possible, it is recommended that application developers be generous in their interpretations of these error codes.

5.2 Conventions for functions returning output in a variable-length buffer

A number of the functions defined in Cryptoki return output produced by some cryptographic mechanism. The amount of output returned by these functions is returned in a variable-length application-supplied buffer. An example of a function of this sort is **C_Encrypt**, which takes some plaintext as an argument, and outputs a buffer full of ciphertext.

These functions have some common calling conventions, which we describe here. Two of the arguments to the function are a pointer to the output buffer (say *pBuf*) and a pointer to a location which will hold the length of the output produced (say *pulBufLen*). There are two ways for an application to call such a function:

1. If *pBuf* is NULL_PTR, then all that the function does is return (in **pulBufLen*) a number of bytes which would suffice to hold the cryptographic output produced from the input to the function. This number may somewhat exceed the precise number of bytes needed, but should not exceed it by a large amount. CKR_OK is returned by the function.
2. If *pBuf* is not NULL_PTR, then **pulBufLen* MUST contain the size in bytes of the buffer pointed to by *pBuf*. If that buffer is large enough to hold the cryptographic output produced from the input to the function, then that cryptographic output is placed there, and CKR_OK is returned by the function and **pulBufLen* is set to the exact number of bytes returned. If the buffer is not large enough, then CKR_BUFFER_TOO_SMALL is returned and **pulBufLen* is set to at least the number of bytes needed to hold the cryptographic output produced from the input to the function.

NOTE: This is a change from previous specs. The problem is that in some decrypt cases, the token doesn't know how big a buffer is needed until the decrypt completes. The act of doing decrypt can mess up the internal encryption state. Many tokens already implement this relaxed behavior, tokens which implement the more precise behavior are still compliant. The one corner case is applications using a token that knows exactly how big the decryption is (through some out of band means), could get CKR_BUFFER_TOO_SMALL returned when it supplied a buffer exactly big enough to hold the decrypted value when it may previously have succeeded.

All functions which use the above convention will explicitly say so.

Cryptographic functions which return output in a variable-length buffer should always return as much output as can be computed from what has been passed in to them thus far. As an example, consider a session which is performing a multiple-part decryption operation with DES in cipher-block chaining mode

2266 with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C_DecryptUpdate**
2267 function. The block size of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether
2268 the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at
2269 least 8 bytes. Hence the call to **C_DecryptUpdate** should return 0 bytes of plaintext. If a single
2270 additional byte of ciphertext is supplied by a subsequent call to **C_DecryptUpdate**, then that call should
2271 return 8 bytes of plaintext (one full DES block).

2272 5.3 Disclaimer concerning sample code

2273 For the remainder of this section, we enumerate the various functions defined in Cryptoki. Most functions
2274 will be shown in use in at least one sample code snippet. For the sake of brevity, sample code will
2275 frequently be somewhat incomplete. In particular, sample code will generally ignore possible error
2276 returns from C library functions, and also will not deal with Cryptoki error returns in a realistic fashion.

2277 5.4 General-purpose functions

2278 Cryptoki provides the following general-purpose functions:

2279 5.4.1 C_Initialize

```
2280 CK_DECLARE_FUNCTION(CK_RV, C_Initialize) {  
2281     CK_VOID_PTR pInitArgs  
2282 };
```

2283 **C_Initialize** initializes the Cryptoki library. *pInitArgs* either has the value **NULL_PTR** or points to a
2284 **CK_C_INITIALIZE_ARGS** structure containing information on how the library should deal with multi-
2285 threaded access. If an application will not be accessing Cryptoki through multiple threads simultaneously,
2286 it can generally supply the value **NULL_PTR** to **C_Initialize** (the consequences of supplying this value will
2287 be explained below).

2288 If *pInitArgs* is non-**NULL_PTR**, **C_Initialize** should cast it to a **CK_C_INITIALIZE_ARGS_PTR** and then
2289 dereference the resulting pointer to obtain the **CK_C_INITIALIZE_ARGS** fields *CreateMutex*,
2290 *DestroyMutex*, *LockMutex*, *UnlockMutex*, *flags*, and *pReserved*. For this version of Cryptoki, the value of
2291 *pReserved* thereby obtained MUST be **NULL_PTR**; if it's not, then **C_Initialize** should return with the
2292 value **CKR_ARGUMENTS_BAD**.

2293 If the **CKF_LIBRARY_CANT_CREATE_OS_THREADS** flag in the *flags* field is set, that indicates that
2294 application threads which are executing calls to the Cryptoki library are not permitted to use the native
2295 operation system calls to spawn off new threads. In other words, the library's code may not create its
2296 own threads. If the library is unable to function properly under this restriction, **C_Initialize** should return
2297 with the value **CKR_NEED_TO_CREATE_THREADS**.

2298 A call to **C_Initialize** specifies one of four different ways to support multi-threaded access via the value of
2299 the **CKF_OS_LOCKING_OK** flag in the *flags* field and the values of the *CreateMutex*, *DestroyMutex*,
2300 *LockMutex*, and *UnlockMutex* function pointer fields:

- 2301 1. If the flag *isn't* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value
2302 **NULL_PTR**), that means that the application *won't* be accessing the Cryptoki library from multiple
2303 threads simultaneously.
- 2304 2. If the flag *is* set, and the function pointer fields *aren't* supplied (*i.e.*, they all have the value
2305 **NULL_PTR**), that means that the application *will* be performing multi-threaded Cryptoki access, and
2306 the library needs to use the native operating system primitives to ensure safe multi-threaded access.
2307 If the library is unable to do this, **C_Initialize** should return with the value **CKR_CANT_LOCK**.
- 2308 3. If the flag *isn't* set, and the function pointer fields *are* supplied (*i.e.*, they all have non-**NULL_PTR**
2309 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the
2310 library needs to use the supplied function pointers for mutex-handling to ensure safe multi-threaded
2311 access. If the library is unable to do this, **C_Initialize** should return with the value
2312 **CKR_CANT_LOCK**.

2313 4. If the flag *is* set, and the function pointer fields are supplied (*i.e.*, they all have non-NULL_PTR
2314 values), that means that the application *will* be performing multi-threaded Cryptoki access, and the
2315 library needs to use either the native operating system primitives or the supplied function pointers for
2316 mutex-handling to ensure safe multi-threaded access. If the library is unable to do this, **C_Initialize**
2317 should return with the value CKR_CANT_LOCK.

2318 If some, but not all, of the supplied function pointers to **C_Initialize** are non-NULL_PTR, then **C_Initialize**
2319 should return with the value CKR_ARGUMENTS_BAD.

2320 A call to **C_Initialize** with *pInitArgs* set to NULL_PTR is treated like a call to **C_Initialize** with *pInitArgs*
2321 pointing to a **CK_C_INITIALIZE_ARGS** which has the *CreateMutex*, *DestroyMutex*, *LockMutex*,
2322 *UnlockMutex*, and *pReserved* fields set to NULL_PTR, and has the *flags* field set to 0.

2323 **C_Initialize** should be the first Cryptoki call made by an application, except for calls to
2324 **C_GetFunctionList**, **C_GetInterfaceList**, or **C_GetInterface**. What this function actually does is
2325 implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or
2326 any other resources it requires.

2327 If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize**
2328 should (eventually) be succeeded by a single call to **C_Finalize**. See [\[PKCS11-UG\]](#) for further details.

2329 Return values: CKR_ARGUMENTS_BAD, CKR_CANT_LOCK,
2330 CKR_CRYPTOKI_ALREADY_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2331 CKR_HOST_MEMORY, CKR_NEED_TO_CREATE_THREADS, CKR_OK.

2332 Example: see **C_GetInfo**.

2333 5.4.2 C_Finalize

```
2334 CK_DECLARE_FUNCTION(CK_RV, C_Finalize) (  
2335     CK_VOID_PTR pReserved  
2336 );
```

2337 **C_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the
2338 last Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for
2339 this version, it should be set to NULL_PTR (if **C_Finalize** is called with a non-NULL_PTR value for
2340 *pReserved*, it should return the value CKR_ARGUMENTS_BAD).

2341 If several applications are using Cryptoki, each one should call **C_Finalize**. Each application's call to
2342 **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application
2343 can make calls to other Cryptoki functions. See [\[PKCS11-UG\]](#) for further details.

2344 *Despite the fact that the parameters supplied to **C_Initialize** can in general allow for safe multi-threaded*
2345 *access to a Cryptoki library, the behavior of **C_Finalize** is nevertheless undefined if it is called by an*
2346 *application while other threads of the application are making Cryptoki calls. The exception to this*
2347 *exceptional behavior of **C_Finalize** occurs when a thread calls **C_Finalize** while another of the*
2348 *application's threads is blocking on Cryptoki's **C_WaitForSlotEvent** function. When this happens, the*
2349 *blocked thread becomes unblocked and returns the value CKR_CRYPTOKI_NOT_INITIALIZED. See*
2350 ***C_WaitForSlotEvent** for more information.*

2351 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2352 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2353 Example: see **C_GetInfo**.

2354 5.4.3 C_GetInfo

```
2355 CK_DECLARE_FUNCTION(CK_RV, C_GetInfo) (  
2356     CK_INFO_PTR pInfo  
2357 );
```

2358 **C_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the
2359 information.

2360 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2361 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2362 Example:

```
2363 CK_INFO info;  
2364 CK_RV rv;  
2365 CK_C_INITIALIZE_ARGS InitArgs;  
2366  
2367 InitArgs.CreateMutex = &MyCreateMutex;  
2368 InitArgs.DestroyMutex = &MyDestroyMutex;  
2369 InitArgs.LockMutex = &MyLockMutex;  
2370 InitArgs.UnlockMutex = &MyUnlockMutex;  
2371 InitArgs.flags = CKF_OS_LOCKING_OK;  
2372 InitArgs.pReserved = NULL_PTR;  
2373  
2374 rv = C_Initialize((CK_VOID_PTR)&InitArgs);  
2375 assert(rv == CKR_OK);  
2376  
2377 rv = C_GetInfo(&info);  
2378 assert(rv == CKR_OK);  
2379 if(info.cryptokiVersion.major == 2) {  
2380     /* Do lots of interesting cryptographic things with the token */  
2381     .  
2382     .  
2383 }  
2384  
2385 rv = C_Finalize(NULL_PTR);  
2386 assert(rv == CKR_OK);
```

2387 5.4.4 C_GetFunctionList

```
2388 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList)(  
2389     CK_FUNCTION_LIST_PTR_PTR ppFunctionList  
2390 );
```

2391 **C_GetFunctionList** obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList*
2392 points to a value which will receive a pointer to the library's **CK_FUNCTION_LIST** structure, which in turn
2393 contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may*
2394 *point into memory which is owned by the Cryptoki library, and which may or may not be writable.*
2395 Whether or not this is the case, no attempt should be made to write to this memory.

2396 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2397 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2398 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2399 Return values: CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2400 CKR_HOST_MEMORY, CKR_OK.

2401 Example:

```
2402 CK_FUNCTION_LIST_PTR pFunctionList;
```



```

2403 CK_C_Initialize pC_Initialize;
2404 CK_RV rv;
2405
2406 /* It's OK to call C_GetFunctionList before calling C_Initialize */
2407 rv = C_GetFunctionList(&pFunctionList);
2408 assert(rv == CKR_OK);
2409 pC_Initialize = pFunctionList -> C_Initialize;
2410
2411 /* Call the C_Initialize function in the library */
2412 rv = (*pC_Initialize)(NULL_PTR);

```

2413 5.4.5 C_GetInterfaceList

```

2414 CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList) (
2415     CK_INTERFACE_PTR      pInterfaceList,
2416     CK_ULONG_PTR          pulCount
2417 );

```

2418 **C_GetInterfaceList** is used to obtain a list of interfaces supported by a Cryptoki library. *pulCount* points
 2419 to the location that receives the number of interfaces.

2420 There are two ways for an application to call **C_GetInterfaceList**:

- 2421 1. If *pInterfaceList* is `NULL_PTR`, then all that **C_GetInterfaceList** does is return (in **pulCount*) the
 2422 number of interfaces, without actually returning a list of interfaces. The contents of **pulCount* on
 2423 entry to **C_GetInterfaceList** has no meaning in this case, and the call returns the value `CKR_OK`.
- 2424 2. If *pInterfaceList* is not `NULL_PTR`, then **pulCount* MUST contain the size (in terms of
 2425 **CK_INTERFACE** elements) of the buffer pointed to by *pInterfaceList*. If that buffer is large enough to
 2426 hold the list of interfaces, then the list is returned in it, and `CKR_OK` is returned. If not, then the call
 2427 to **C_GetInterfaceList** returns the value `CKR_BUFFER_TOO_SMALL`. In either case, the value
 2428 **pulCount* is set to hold the number of interfaces.

2429 Because **C_GetInterfaceList** does not allocate any space of its own, an application will often call
 2430 **C_GetInterfaceList** twice. However, this behavior is by no means required.

2431 **C_GetInterfaceList** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
 2432 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
 2433 *library, and which may or may not be writable.* Whether or not this is the case, no attempt should be
 2434 made to write to this memory. The same caveat applies to the interface names returned.

2435
 2436 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
 2437 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
 2438 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2439 Return values: `CKR_BUFFER_TOO_SMALL`, `CKR_ARGUMENTS_BAD`, `CKR_FUNCTION_FAILED`,
 2440 `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`.

2441 Example:

```

2442 CK_ULONG ulCount=0;
2443 CK_INTERFACE_PTR interfaceList=NULL;
2444 CK_RV rv;
2445 int I;
2446
2447 /* get number of interfaces */

```

```

2448 rv = C_GetInterfaceList(NULL, &ulCount);
2449 if (rv == CKR_OK) {
2450     /* get copy of interfaces */
2451     interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
2452     rv = C_GetInterfaceList(interfaceList, &ulCount);
2453     for(i=0; i<ulCount; i++) {
2454         printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2455             interfaceList[i].pInterfaceName,
2456             ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
2457             ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
2458             interfaceList[i].pFunctionList,
2459             interfaceList[i].flags);
2460     }
2461 }
2462

```

2463 5.4.6 C_GetInterface

```

2464 CK_DECLARE_FUNCTION(CK_RV, C_GetInterface) (
2465     CK_UTF8CHAR_PTR      pInterfaceName,
2466     CK_VERSION_PTR       pVersion,
2467     CK_INTERFACE_PTR_PTR ppInterface,
2468     CK_FLAGS              flags
2469 );

```

2470 **C_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies
2471 the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that
2472 receives the interface, *flags* specifies the required interface flags.

2473 There are multiple ways for an application to specify a particular interface when calling **C_GetInterface**:

- 2474 1. If *pInterfaceName* is not NULL_PTR, the name of the interface returned must match. If
2475 *pInterfaceName* is NULL_PTR, the cryptoki library can return a default interface of its choice
- 2476 2. If *pVersion* is not NULL_PTR, the version of the interface returned must match. If *pVersion* is
2477 NULL_PTR, the cryptoki library can return an interface of any version
- 2478 3. If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include
2479 additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

2480 **C_GetInterface** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
2481 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
2482 *library, and which may or may not be writable.* Whether or not this is the case, no attempt should be
2483 made to write to this memory. The same caveat applies to the interface names returned.

2484 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
2485 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
2486 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

2487 Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED,
2488 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

2489 Example:

```

2490 CK_INTERFACE_PTR interface;
2491 CK_RV rv;

```

```

2492 CK_VERSION version;
2493 CK_FLAGS flags=CKF_ INTERFACE_FORK_SAFE;
2494
2495 /* get default interface */
2496 rv = C_GetInterface(NULL,NULL,&interface,flags);
2497 if (rv == CKR_OK) {
2498     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2499         interface->pInterfaceName,
2500         ((CK_VERSION *)interface->pFunctionList)->major,
2501         ((CK_VERSION *)interface->pFunctionList)->minor,
2502         interface->pFunctionList,
2503         interface->flags);
2504 }
2505
2506 /* get default standard interface */
2507 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",NULL,&interface,flags);
2508 if (rv == CKR_OK) {
2509     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
2510         interface->pInterfaceName,
2511         ((CK_VERSION *)interface->pFunctionList)->major,
2512         ((CK_VERSION *)interface->pFunctionList)->minor,
2513         interface->pFunctionList,
2514         interface->flags);
2515 }
2516
2517 /* get specific standard version interface */
2518 version.major=3;
2519 version.minor=0;
2520 rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",&version,&interface,flags);
2521 if (rv == CKR_OK) {
2522     CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
2523
2524     /* ... use the new functions */
2525     pkcs11->C_LoginUser(hSession,userType,pPin,ulPinLen,
2526                         pUsername,ulUsernameLen);
2527 }
2528
2529 /* get specific vendor version interface */
2530 version.major=1;
2531 version.minor=0;
2532 rv = C_GetInterface((CK_UTF8CHAR_PTR)
2533     "Vendor VendorName",&version,&interface,flags);
2534 if (rv == CKR_OK) {

```

```

2535     CK_FUNCTION_LIST_VENDOR_1_0_PTR pkcs11=interface->pFunctionList;
2536
2537     /* ... use vendor specific functions */
2538     pkcs11->C_VendorFunction1(param1,param2,param3);
2539 }
2540

```

2541 5.5 Slot and token management functions

2542 Cryptoki provides the following functions for slot and token management:

2543 5.5.1 C_GetSlotList

```

2544 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotList) (
2545     CK_BBOOL tokenPresent,
2546     CK_SLOT_ID_PTR pSlotList,
2547     CK_ULONG_PTR pulCount
2548 );

```

2549 **C_GetSlotList** is used to obtain a list of slots in the system. *tokenPresent* indicates whether the list
2550 obtained includes only those slots with a token present (CK_TRUE), or all slots (CK_FALSE); *pulCount*
2551 points to the location that receives the number of slots.

2552 There are two ways for an application to call **C_GetSlotList**:

- 2553 1. If *pSlotList* is NULL_PTR, then all that **C_GetSlotList** does is return (in **pulCount*) the number of
2554 slots, without actually returning a list of slots. The contents of the buffer pointed to by *pulCount* on
2555 entry to **C_GetSlotList** has no meaning in this case, and the call returns the value CKR_OK.
- 2556 2. If *pSlotList* is not NULL_PTR, then **pulCount* MUST contain the size (in terms of **CK_SLOT_ID**
2557 elements) of the buffer pointed to by *pSlotList*. If that buffer is large enough to hold the list of slots,
2558 then the list is returned in it, and CKR_OK is returned. If not, then the call to **C_GetSlotList** returns
2559 the value CKR_BUFFER_TOO_SMALL. In either case, the value **pulCount* is set to hold the number
2560 of slots.

2561 Because **C_GetSlotList** does not allocate any space of its own, an application will often call
2562 **C_GetSlotList** twice (or sometimes even more times—if an application is trying to get a list of all slots
2563 with a token present, then the number of such slots can (unfortunately) change between when the
2564 application asks for how many such slots there are and when the application asks for the slots
2565 themselves). However, multiple calls to **C_GetSlotList** are by no means *required*.

2566 All slots which **C_GetSlotList** reports MUST be able to be queried as valid slots by **C_GetSlotInfo**.
2567 Furthermore, the set of slots accessible through a Cryptoki library is checked at the time that
2568 **C_GetSlotList**, for list length prediction (NULL pSlotList argument) is called. If an application calls
2569 **C_GetSlotList** with a non-NULL pSlotList, and *then* the user adds or removes a hardware device, the
2570 changed slot list will only be visible and effective if **C_GetSlotList** is called again with NULL. Even if **C_**
2571 **GetSlotList** is successfully called this way, it may or may not be the case that the changed slot list will be
2572 successfully recognized depending on the library implementation. On some platforms, or earlier PKCS11
2573 compliant libraries, it may be necessary to successfully call **C_Initialize** or to restart the entire system.

2574

2575 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
2576 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2577 CKR_HOST_MEMORY, CKR_OK.

2578 Example:

```

2579 CK_ULONG ulSlotCount, ulSlotWithTokenCount;
2580 CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList;

```

```

2581 CK_RV rv;
2582
2583 /* Get list of all slots */
2584 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulSlotCount);
2585 if (rv == CKR_OK) {
2586     pSlotList =
2587         (CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
2588     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
2589     if (rv == CKR_OK) {
2590         /* Now use that list of all slots */
2591         .
2592         .
2593     }
2594
2595     free(pSlotList);
2596 }
2597
2598 /* Get list of all slots with a token present */
2599 pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
2600 ulSlotWithTokenCount = 0;
2601 while (1) {
2602     rv = C_GetSlotList(
2603         CK_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);
2604     if (rv != CKR_BUFFER_TOO_SMALL)
2605         break;
2606     pSlotWithTokenList = realloc(
2607         pSlotWithTokenList,
2608         ulSlotWithTokenList*sizeof(CK_SLOT_ID));
2609 }
2610
2611 if (rv == CKR_OK) {
2612     /* Now use that list of all slots with a token present */
2613     .
2614     .
2615 }
2616
2617 free(pSlotWithTokenList);

```

2618 5.5.2 C_GetSlotInfo

```

2619 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo)(
2620     CK_SLOT_ID slotID,
2621     CK_SLOT_INFO_PTR pInfo
2622 );

```

2623 **C_GetSlotInfo** obtains information about a particular slot in the system. *slotID* is the ID of the slot; *pInfo*
 2624 points to the location that receives the slot information.

2625 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 2626 CKR_DEVICE_ERROR, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
 2627 CKR_OK, CKR_SLOT_ID_INVALID.

2628 Example: see **C_GetTokenInfo**.

2629 5.5.3 C_GetTokenInfo

```
2630 CK_DECLARE_FUNCTION(CK_RV, C_GetTokenInfo) (
2631     CK_SLOT_ID slotID,
2632     CK_TOKEN_INFO_PTR pInfo
2633 );
```

2634 **C_GetTokenInfo** obtains information about a particular token in the system. *slotID* is the ID of the
 2635 token's slot; *pInfo* points to the location that receives the token information.

2636 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 2637 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 2638 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT,
 2639 CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

2640 Example:

```
2641 CK_ULONG ulCount;
2642 CK_SLOT_ID_PTR pSlotList;
2643 CK_SLOT_INFO slotInfo;
2644 CK_TOKEN_INFO tokenInfo;
2645 CK_RV rv;
2646
2647 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulCount);
2648 if ((rv == CKR_OK) && (ulCount > 0)) {
2649     pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));
2650     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulCount);
2651     assert(rv == CKR_OK);
2652
2653     /* Get slot information for first slot */
2654     rv = C_GetSlotInfo(pSlotList[0], &slotInfo);
2655     assert(rv == CKR_OK);
2656
2657     /* Get token information for first slot */
2658     rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);
2659     if (rv == CKR_TOKEN_NOT_PRESENT) {
2660         .
2661         .
2662     }
2663     .
2664     .
2665     free(pSlotList);
```

2666 }

2667 5.5.4 C_WaitForSlotEvent

```
2668 CK_DECLARE_FUNCTION(CK_RV, C_WaitForSlotEvent) (  
2669     CK_FLAGS flags,  
2670     CK_SLOT_ID_PTR pSlot,  
2671     CK_VOID_PTR pReserved  
2672 );
```

2673 **C_WaitForSlotEvent** waits for a slot event, such as token insertion or token removal, to occur. *flags*
2674 determines whether or not the **C_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot*
2675 points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved
2676 for future versions; for this version of Cryptoki, it should be NULL_PTR.

2677 At present, the only flag defined for use in the *flags* argument is **CKF_DONT_BLOCK**:

2678 Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any
2679 unrecognized events involving that slot have occurred. When an application initially calls **C_Initialize**,
2680 every slot's event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in
2681 which the event occurred is set.

2682 If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and some
2683 slot's event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the
2684 location pointed to by *pSlot*. If more than one slot's event flag is set at the time of the call, one such slot
2685 is chosen by the library to have its event flag cleared and to have its slot ID returned.

2686 If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and no
2687 slot's event flag is set, then the call returns with the value CKR_NO_EVENT. In this case, the contents of
2688 the location pointed to by *pSlot* when **C_WaitForSlotEvent** are undefined.

2689 If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag clear in the *flags* argument, then the
2690 call behaves as above, except that it will block. That is, if no slot's event flag is set at the time of the call,
2691 **C_WaitForSlotEvent** will wait until some slot's event flag becomes set. If a thread of an application has
2692 a **C_WaitForSlotEvent** call blocking when another thread of that application calls **C_Finalize**, the
2693 **C_WaitForSlotEvent** call returns with the value CKR_CRYPTOKI_NOT_INITIALIZED.

2694 *Although the parameters supplied to C_Initialize can in general allow for safe multi-threaded access to a*
2695 *Cryptoki library, C_WaitForSlotEvent is exceptional in that the behavior of Cryptoki is undefined if*
2696 *multiple threads of a single application make simultaneous calls to C_WaitForSlotEvent.*

2697 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
2698 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_NO_EVENT,
2699 CKR_OK.

2700 Example:

```
2701 CK_FLAGS flags = 0;  
2702 CK_SLOT_ID slotID;  
2703 CK_SLOT_INFO slotInfo;  
2704 CK_RV rv;  
2705 .  
2706 .  
2707 /* Block and wait for a slot event */  
2708 rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);  
2709 assert(rv == CKR_OK);  
2710  
2711 /* See what's up with that slot */
```

```

2712 rv = C_GetSlotInfo(slotID, &slotInfo);
2713 assert(rv == CKR_OK);
2714

```

2715 5.5.5 C_GetMechanismList

```

2716 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList) (
2717     CK_SLOT_ID slotID,
2718     CK_MECHANISM_TYPE_PTR pMechanismList,
2719     CK_ULONG_PTR pulCount
2720 );

```

2721 **C_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID
 2722 of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

2723 There are two ways for an application to call **C_GetMechanismList**:

- 2724 1. If *pMechanismList* is NULL_PTR, then all that **C_GetMechanismList** does is return (in **pulCount*)
 2725 the number of mechanisms, without actually returning a list of mechanisms. The contents of
 2726 **pulCount* on entry to **C_GetMechanismList** has no meaning in this case, and the call returns the
 2727 value CKR_OK.
- 2728 2. If *pMechanismList* is not NULL_PTR, then **pulCount* MUST contain the size (in terms of
 2729 **CK_MECHANISM_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large
 2730 enough to hold the list of mechanisms, then the list is returned in it, and CKR_OK is returned. If not,
 2731 then the call to **C_GetMechanismList** returns the value CKR_BUFFER_TOO_SMALL. In either
 2732 case, the value **pulCount* is set to hold the number of mechanisms.

2733 Because **C_GetMechanismList** does not allocate any space of its own, an application will often call
 2734 **C_GetMechanismList** twice. However, this behavior is by no means required.

2735 Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
 2736 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 2737 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
 2738 CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED,
 2739 CKR_ARGUMENTS_BAD.

2740 Example:

```

2741 CK_SLOT_ID slotID;
2742 CK_ULONG ulCount;
2743 CK_MECHANISM_TYPE_PTR pMechanismList;
2744 CK_RV rv;
2745
2746 .
2747 .
2748 rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);
2749 if ((rv == CKR_OK) && (ulCount > 0)) {
2750     pMechanismList =
2751         (CK_MECHANISM_TYPE_PTR)
2752         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));
2753     rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);
2754     if (rv == CKR_OK) {
2755         .
2756         .

```



```

2757     }
2758     free(pMechanismList);
2759 }

```

2760 5.5.6 C_GetMechanismInfo

```

2761 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo) (
2762     CK_SLOT_ID slotID,
2763     CK_MECHANISM_TYPE type,
2764     CK_MECHANISM_INFO_PTR pInfo
2765 );

```

2766 **C_GetMechanismInfo** obtains information about a particular mechanism possibly supported by a token.
2767 *slotID* is the ID of the token's slot; *type* is the type of mechanism; *pInfo* points to the location that receives
2768 the mechanism information.

2769 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2770 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2771 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID,
2772 CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

2773 Example:

```

2774 CK_SLOT_ID slotID;
2775 CK_MECHANISM_INFO info;
2776 CK_RV rv;
2777
2778 .
2779 .
2780 /* Get information about the CKM_MD2 mechanism for this token */
2781 rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
2782 if (rv == CKR_OK) {
2783     if (info.flags & CKF_DIGEST) {
2784         .
2785         .
2786     }
2787 }

```

2788 5.5.7 C_InitToken

```

2789 CK_DECLARE_FUNCTION(CK_RV, C_InitToken) (
2790     CK_SLOT_ID slotID,
2791     CK_UTF8CHAR_PTR pPin,
2792     CK_ULONG ulPinLen,
2793     CK_UTF8CHAR_PTR pLabel
2794 );

```

2795 **C_InitToken** initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN
2796 (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-
2797 byte label of the token (which MUST be padded with blank characters, and which MUST *not* be null-

2798 terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may
 2799 impose subset restrictions.

2800 If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the
 2801 initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the
 2802 existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after
 2803 the function completes successfully. If the SO PIN is lost, then the card **MUST** be reinitialized using a
 2804 mechanism outside the scope of this standard. The **CKF_TOKEN_INITIALIZED** flag in the
 2805 **CK_TOKEN_INFO** structure indicates the action that will result from calling **C_InitToken**. If set, the token
 2806 will be reinitialized, and the client **MUST** supply the existing SO password in *pPin*.

2807 When a token is initialized, all objects that can be destroyed are destroyed (i.e., all except for
 2808 “indestructible” objects such as keys built into the token). Also, access by the normal user is disabled
 2809 until the SO sets the normal user’s PIN. Depending on the token, some “default” objects may be created,
 2810 and attributes of some objects may be set to default values.

2811 If the token has a “protected authentication path”, as indicated by the
 2812 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
 2813 that there is some way for a user to be authenticated to the token without having the application send a
 2814 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the
 2815 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin*
 2816 parameter to **C_InitToken** should be **NULL_PTR**. During the execution of **C_InitToken**, the SO’s PIN will
 2817 be entered through the protected authentication path.

2818 If the token has a protected authentication path other than a PINpad, then it is token-dependent whether
 2819 or not **C_InitToken** can be used to initialize the token.

2820 A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a
 2821 call to **C_InitToken** is made under such circumstances, the call fails with error **CKR_SESSION_EXISTS**.
 2822 Unfortunately, it may happen when **C_InitToken** is called that some other application *does* have an open
 2823 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other
 2824 applications using the token. If this is the case, then the consequences of the **C_InitToken** call are
 2825 undefined.

2826 The **C_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations,
 2827 an initialization mechanism outside the scope of Cryptoki **MUST** be employed. The definition of “complex
 2828 token” is product specific.

2829 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
 2830 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
 2831 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_PIN_INCORRECT**,
 2832 **CKR_PIN_LOCKED**, **CKR_SESSION_EXISTS**, **CKR_SLOT_ID_INVALID**,
 2833 **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
 2834 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

2835 Example:

```

2836 CK_SLOT_ID slotID;
2837 CK_UTF8CHAR pin[] = {"MyPIN"};
2838 CK_UTF8CHAR label[32];
2839 CK_RV rv;
2840
2841 .
2842 .
2843 memset(label, '\0', sizeof(label));
2844 memcpy(label, "My first token", strlen("My first token"));
2845 rv = C_InitToken(slotID, pin, strlen(pin), label);
2846 if (rv == CKR_OK) {
2847     .
  
```

2848 .
2849 }

2850 5.5.8 C_InitPIN

```
2851 CK_DECLARE_FUNCTION(CK_RV, C_InitPIN) (  
2852     CK_SESSION_HANDLE hSession,  
2853     CK_UTF8CHAR_PTR pPin,  
2854     CK_ULONG ulPinLen  
2855 );
```

2856 **C_InitPIN** initializes the normal user's PIN. *hSession* is the session's handle; *pPin* points to the normal
2857 user's PIN; *ulPinLen* is the length in bytes of the PIN. This standard allows PIN values to contain any
2858 valid UTF8 character, but the token may impose subset restrictions.

2859 **C_InitPIN** can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any
2860 other state fails with error CKR_USER_NOT_LOGGED_IN.

2861 If the token has a "protected authentication path", as indicated by the
2862 CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means
2863 that there is some way for a user to be authenticated to the token without having to send a PIN through
2864 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
2865 on the slot device. To initialize the normal user's PIN on a token with such a protected authentication
2866 path, the *pPin* parameter to **C_InitPIN** should be NULL_PTR. During the execution of **C_InitPIN**, the SO
2867 will enter the new PIN through the protected authentication path.

2868 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
2869 or not **C_InitPIN** can be used to initialize the normal user's token access.

2870 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2871 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
2872 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INVALID,
2873 CKR_PIN_LEN_RANGE, CKR_SESSION_CLOSED, CKR_SESSION_READ_ONLY,
2874 CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_WRITE_PROTECTED,
2875 CKR_USER_NOT_LOGGED_IN, CKR_ARGUMENTS_BAD.

2876 Example:

```
2877 CK_SESSION_HANDLE hSession;  
2878 CK_UTF8CHAR newPin[] = {"NewPIN"};  
2879 CK_RV rv;  
2880  
2881 rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);  
2882 if (rv == CKR_OK) {  
2883     .  
2884     .  
2885 }
```

2886 5.5.9 C_SetPIN

```
2887 CK_DECLARE_FUNCTION(CK_RV, C_SetPIN) (  
2888     CK_SESSION_HANDLE hSession,  
2889     CK_UTF8CHAR_PTR pOldPin,  
2890     CK_ULONG ulOldLen,  
2891     CK_UTF8CHAR_PTR pNewPin,  
2892     CK_ULONG ulNewLen  
2893 );
```

2894 **C_SetPIN** modifies the PIN of the user that is currently logged in, or the CKU_USER PIN if the session is
2895 not logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in
2896 bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This
2897 standard allows PIN values to contain any valid UTF8 character, but the token may impose subset
2898 restrictions.

2899 **C_SetPIN** can only be called in the "R/W Public Session" state, "R/W SO Functions" state, or "R/W User
2900 Functions" state. An attempt to call it from a session in any other state fails with error
2901 CKR_SESSION_READ_ONLY.

2902 If the token has a "protected authentication path", as indicated by the
2903 CKF_PROTECTED_AUTHENTICATION_PATH flag in its **CK_TOKEN_INFO** being set, then that means
2904 that there is some way for a user to be authenticated to the token without having to send a PIN through
2905 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
2906 on the slot device. To modify the current user's PIN on a token with such a protected authentication path,
2907 the *pOldPin* and *pNewPin* parameters to **C_SetPIN** should be NULL_PTR. During the execution of
2908 **C_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication
2909 path. It is not specified how the PIN pad should be used to enter *two* PINs; this varies.

2910 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
2911 or not **C_SetPIN** can be used to modify the current user's PIN.

2912 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2913 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
2914 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_INCORRECT,
2915 CKR_PIN_INVALID, CKR_PIN_LEN_RANGE, CKR_PIN_LOCKED, CKR_SESSION_CLOSED,
2916 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
2917 CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

2918 Example:

```
2919 CK_SESSION_HANDLE hSession;  
2920 CK_UTF8CHAR oldPin[] = {"OldPIN"};  
2921 CK_UTF8CHAR newPin[] = {"NewPIN"};  
2922 CK_RV rv;  
2923  
2924 rv = C_SetPIN(  
2925     hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);  
2926 if (rv == CKR_OK) {  
2927     .  
2928     .  
2929 }
```

2930 5.6 Session management functions

2931 A typical application might perform the following series of steps to make use of a token (note that there
2932 are other reasonable sequences of events that an application might perform):

- 2933 1. Select a token.
- 2934 2. Make one or more calls to **C_OpenSession** to obtain one or more sessions with the token.
- 2935 3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a
2936 shared login state, **C_Login** only needs to be called for one of the sessions.
- 2937 4. Perform cryptographic operations using the sessions with the token.
- 2938 5. Call **C_CloseSession** once for each session that the application has with the token, or call
2939 **C_CloseAllSessions** to close all the application's sessions simultaneously.

2940 As has been observed, an application may have concurrent sessions with more than one token. It is also
2941 possible for a token to have concurrent sessions with more than one application.
2942 Cryptoki provides the following functions for session management:

2943 5.6.1 C_OpenSession

```
2944 CK_DECLARE_FUNCTION(CK_RV, C_OpenSession) (  
2945     CK_SLOT_ID slotID,  
2946     CK_FLAGS flags,  
2947     CK_VOID_PTR pApplication,  
2948     CK_NOTIFY Notify,  
2949     CK_SESSION_HANDLE_PTR phSession  
2950 );
```

2951 **C_OpenSession** opens a session between an application and a token in a particular slot. *slotID* is the
2952 slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to
2953 the notification callback; *Notify* is the address of the notification callback function (see Section 5.21);
2954 *phSession* points to the location that receives the handle for the new session.

2955 When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or
2956 more bit flags defined in the **CK_SESSION_INFO** data type. For legacy reasons, the
2957 **CKF_SERIAL_SESSION** bit MUST always be set; if a call to **C_OpenSession** does not have this bit set,
2958 the call should return unsuccessfully with the error code
2959 CKR_SESSION_PARALLEL_NOT_SUPPORTED.

2960 There may be a limit on the number of concurrent sessions an application may have with the token, which
2961 may depend on whether the session is "read-only" or "read/write". An attempt to open a session which
2962 does not succeed because there are too many existing sessions of some type should return
2963 CKR_SESSION_COUNT.

2964 If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only read-only
2965 sessions may be opened with it.

2966 If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any
2967 attempt to open a R/O session with the token fails with error code
2968 CKR_SESSION_READ_WRITE_SO_EXISTS (see [PKCS11-UG] for further details).

2969 The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the
2970 application does not wish to support callbacks, it should pass a value of NULL_PTR as the *Notify*
2971 parameter. See Section 5.21 for more information about application callbacks.

2972 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2973 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
2974 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_COUNT,
2975 CKR_SESSION_PARALLEL_NOT_SUPPORTED, CKR_SESSION_READ_WRITE_SO_EXISTS,
2976 CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED,
2977 CKR_TOKEN_WRITE_PROTECTED, CKR_ARGUMENTS_BAD.

2978 Example: see **C_CloseSession**.

2979 5.6.2 C_CloseSession

```
2980 CK_DECLARE_FUNCTION(CK_RV, C_CloseSession) (  
2981     CK_SESSION_HANDLE hSession  
2982 );
```

2983 **C_CloseSession** closes a session between an application and a token. *hSession* is the session's
2984 handle.

2985 When a session is closed, all session objects created by the session are destroyed automatically, even if
2986 the application has other sessions "using" the objects (see [PKCS11-UG] for further details).

2987 If this function is successful and it closes the last session between the application and the token, the login
2988 state of the token for the application returns to public sessions. Any new sessions to the token opened by
2989 the application will be either R/O Public or R/W Public sessions.

2990 Depending on the token, when the last open session any application has with the token is closed, the
2991 token may be “ejected” from its reader (if this capability exists).

2992 Despite the fact this **C_CloseSession** is supposed to close a session, the return value
2993 CKR_SESSION_CLOSED is an *error* return. It actually indicates the (probably somewhat unlikely) event
2994 that while this function call was executing, another call was made to **C_CloseSession** to close this
2995 particular session, and that call finished executing first. Such uses of sessions are a bad idea, and
2996 Cryptoki makes little promise of what will occur in general if an application indulges in this sort of
2997 behavior.

2998 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
2999 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3000 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3001 Example:

```
3002 CK_SLOT_ID slotID;  
3003 CK_BYTE application;  
3004 CK_NOTIFY MyNotify;  
3005 CK_SESSION_HANDLE hSession;  
3006 CK_RV rv;  
3007  
3008 .  
3009 .  
3010 application = 17;  
3011 MyNotify = &EncryptionSessionCallback;  
3012 rv = C_OpenSession(  
3013     slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,  
3014     (CK_VOID_PTR) &application, MyNotify,  
3015     &hSession);  
3016 if (rv == CKR_OK) {  
3017     .  
3018     .  
3019     C_CloseSession(hSession);  
3020 }
```

3021 5.6.3 C_CloseAllSessions

```
3022 CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions) (  
3023     CK_SLOT_ID slotID  
3024 );
```

3025 **C_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token's slot.
3026 When a session is closed, all session objects created by the session are destroyed automatically.
3027 After successful execution of this function, the login state of the token for the application returns to public
3028 sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W
3029 Public sessions.

3030 Depending on the token, when the last open session any application has with the token is closed, the
3031 token may be “ejected” from its reader (if this capability exists).

3032 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3033 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3034 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

3035 Example:

```
3036 CK_SLOT_ID slotID;  
3037 CK_RV rv;  
3038  
3039 .  
3040 .  
3041 rv = C_CloseAllSessions(slotID);
```

3042 5.6.4 C_GetSessionInfo

```
3043 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo) (  
3044     CK_SESSION_HANDLE hSession,  
3045     CK_SESSION_INFO_PTR pInfo  
3046 );
```

3047 **C_GetSessionInfo** obtains information about a session. *hSession* is the session's handle; *pInfo* points to
3048 the location that receives the session information.

3049 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3050 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3051 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3052 CKR_ARGUMENTS_BAD.

3053 Example:

```
3054 CK_SESSION_HANDLE hSession;  
3055 CK_SESSION_INFO info;  
3056 CK_RV rv;  
3057  
3058 .  
3059 .  
3060 rv = C_GetSessionInfo(hSession, &info);  
3061 if (rv == CKR_OK) {  
3062     if (info.state == CKS_RW_USER_FUNCTIONS) {  
3063         .  
3064         .  
3065     }  
3066     .  
3067     .  
3068 }
```

3069 5.6.5 C_SessionCancel

```
3070 CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel) (  
3071     CK_SESSION_HANDLE hSession  
3072     CK_FLAGS flags  
3073 );
```

3074 **C_SessionCancel** terminates active session based operations. *hSession* is the session's handle; *flags*
 3075 indicates the operations to cancel.

3076 To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical
 3077 bitwise OR of one or more of the bit flags defined in the **CK_MECHANISM_INFO** structure.

3078 If no flags are set, the session state will not be modified and CKR_OK will be returned.

3079 If a flag is set for an operation that has not been initialized in the session, the operation flag will be
 3080 ignored and **C_SessionCancel** will behave as if the operation flag was not set.

3081 If any of the operations indicated by the *flags* parameter cannot be cancelled,
 3082 CKR_OPERATION_CANCEL_FAILED must be returned. If multiple operation flags were set and
 3083 CKR_OPERATION_CANCEL_FAILED is returned, this function does not provide any information about
 3084 which operation(s) could not be cancelled. If an application desires to know if any single operation could
 3085 not be cancelled, the application should not call **C_SessionCancel** with multiple flags set.

3086 If **C_SessionCancel** is called from an application callback (see Section 5.21), no action will be taken by
 3087 the library and CKR_FUNCTION_FAILED must be returned.

3088 If **C_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation
 3089 should still be left in an active state. However, it is expected that some Cryptoki implementations may not
 3090 support this and return CKR_OPERATION_CANCEL_FAILED unless flags for both operations are
 3091 provided.

3092 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 3093 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 3094 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_CANCEL_FAILED,
 3095 CKR_TOKEN_NOT_PRESENT.

3096 Example:

```

3097 CK_SESSION_HANDLE hSession;
3098 CK_RV rv;
3099
3100 rv = C_EncryptInit(hSession, &mechanism, hKey);
3101 if (rv != CKR_OK)
3102 {
3103     .
3104     .
3105 }
3106
3107 rv = C_SessionCancel (hSession, CKF_ENCRYPT);
3108 if (rv != CKR_OK)
3109 {
3110     .
3111     .
3112 }
3113
3114 rv = C_EncryptInit(hSession, &mechanism, hKey);
3115 if (rv != CKR_OK)
3116 {
3117     .
3118     .
3119 }
  
```


3120

3121

3122

3123

3124 Below are modifications to existing API descriptions to allow an alternate method of cancelling individual
3125 operations. The additional text is highlighted.

3126 5.6.6 C_GetOperationState

```
3127 CK_DECLARE_FUNCTION(CK_RV, C_GetOperationState) (  
3128     CK_SESSION_HANDLE hSession,  
3129     CK_BYTE_PTR pOperationState,  
3130     CK_ULONG_PTR pulOperationStateLen  
3131 );
```

3132 **C_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a
3133 string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the
3134 state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

3135 Although the saved state output by **C_GetOperationState** is not really produced by a "cryptographic
3136 mechanism", **C_GetOperationState** nonetheless uses the convention described in Section 5.2 on
3137 producing output.

3138 Precisely what the "cryptographic operations state" this function saves is varies from token to token;
3139 however, this state is what is provided as input to **C_SetOperationState** to restore the cryptographic
3140 activities of a session.

3141 Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is
3142 using the **CKM_SHA_1** mechanism). Suppose that the message digest operation was initialized
3143 properly, and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The
3144 application now wants to "save the state" of this digest operation, so that it can continue it later. In this
3145 particular case, since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic
3146 operations state of the session most likely consists of three distinct parts: the state of SHA-1's 160-bit
3147 internal chaining variable; the 16 bytes of unprocessed input data; and some administrative data
3148 indicating that this saved state comes from a session which was performing SHA-1 hashing. Taken
3149 together, these three pieces of information suffice to continue the current hashing operation at a later
3150 time.

3151 Consider next a session which is performing an encryption operation with DES (a block cipher with a
3152 block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM_DES_CBC**
3153 mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been
3154 supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already
3155 been produced and output. In this case, the cryptographic operations state of the session most likely
3156 consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-
3157 block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some
3158 administrative data indicating that this saved state comes from a session which was performing DES
3159 encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState**
3160 for more information on whether or not the key is present in the saved state).

3161 If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the
3162 cryptographic operations state of the session will contain all the necessary information to restore both
3163 operations.

3164 An attempt to save the cryptographic operations state of a session which does not currently have some
3165 active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message
3166 recovery, verification without message recovery, or some legal combination of two of these) should fail
3167 with the error **CKR_OPERATION_NOT_INITIALIZED**.

3168 An attempt to save the cryptographic operations state of a session which is performing an appropriate
3169 cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain
3170 necessary state information and/or key information can't leave the token, for example) should fail with the
3171 error **CKR_STATE_UNSAVEABLE**.

3172 Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
3173 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3174 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3175 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3176 CKR_STATE_UNSAVEABLE, CKR_ARGUMENTS_BAD.

3177 Example: see **C_SetOperationState**.

3178 5.6.7 C_SetOperationState

```
3179 CK_DECLARE_FUNCTION(CK_RV, C_SetOperationState)(  
3180     CK_SESSION_HANDLE hSession,  
3181     CK_BYTE_PTR pOperationState,  
3182     CK_ULONG ulOperationStateLen,  
3183     CK_OBJECT_HANDLE hEncryptionKey,  
3184     CK_OBJECT_HANDLE hAuthenticationKey  
3185 );
```

3186 **C_SetOperationState** restores the cryptographic operations state of a session from a string of bytes
3187 obtained with **C_GetOperationState**. *hSession* is the session's handle; *pOperationState* points to the
3188 location holding the saved state; *ulOperationStateLen* holds the length of the saved state;
3189 *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption
3190 operation in the restored session (or 0 if no encryption or decryption key is needed, either because no
3191 such operation is ongoing in the stored session or because all the necessary key information is present in
3192 the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing
3193 signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either
3194 because no such operation is ongoing in the stored session or because all the necessary key information
3195 is present in the saved state).

3196 The state need not have been obtained from the same session (the "source session") as it is being
3197 restored to (the "destination session"). However, the source session and destination session should have
3198 a common session state (e.g., CKS_RW_USER_FUNCTIONS), and should be with a common token.
3199 There is also no guarantee that cryptographic operations state may be carried across logins, or across
3200 different Cryptoki implementations.

3201 If **C_SetOperationState** is supplied with alleged saved cryptographic operations state which it can
3202 determine is not valid saved state (or is cryptographic operations state from a session with a different
3203 session state, or is cryptographic operations state from a different token), it fails with the error
3204 CKR_SAVED_STATE_INVALID.

3205 Saved state obtained from calls to **C_GetOperationState** may or may not contain information about keys
3206 in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing
3207 encryption or decryption operation, and the key in use for the operation is not saved in the state, then it
3208 MUST be supplied to **C_SetOperationState** in the *hEncryptionKey* argument. If it is not, then
3209 **C_SetOperationState** will fail and return the error CKR_KEY_NEEDED. If the key in use for the
3210 operation is saved in the state, then it *can* be supplied in the *hEncryptionKey* argument, but this is not
3211 required.

3212 Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification
3213 operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to
3214 **C_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C_SetOperationState** will
3215 fail with the error CKR_KEY_NEEDED. If the key in use for the operation is saved in the state, then it *can*
3216 be supplied in the *hAuthenticationKey* argument, but this is not required.

3217 If an *irrelevant* key is supplied to **C_SetOperationState** call (e.g., a nonzero key handle is submitted in
3218 the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an
3219 ongoing encryption or decryption operation, then **C_SetOperationState** fails with the error
3220 CKR_KEY_NOT_NEEDED.

3221 If a key is supplied as an argument to **C_SetOperationState**, and **C_SetOperationState** can somehow
3222 detect that this key was not the key being used in the source session for the supplied cryptographic

3223 operations state (it may be able to detect this if the key or a hash of the key is present in the saved state,
3224 for example), then **C_SetOperationState** fails with the error CKR_KEY_CHANGED.

3225 An application can look at the **CKF_RESTORE_KEY_NOT_NEEDED** flag in the flags field of the
3226 **CK_TOKEN_INFO** field for a token to determine whether or not it needs to supply key handles to
3227 **C_SetOperationState** calls. If this flag is true, then a call to **C_SetOperationState** *never* needs a key
3228 handle to be supplied to it. If this flag is false, then at least some of the time, **C_SetOperationState**
3229 requires a key handle, and so the application should probably *always* pass in any relevant key handles
3230 when restoring cryptographic operations state to a session.

3231 **C_SetOperationState** can successfully restore cryptographic operations state to a session even if that
3232 session has active cryptographic or object search operations when **C_SetOperationState** is called (the
3233 ongoing operations are abruptly cancelled).

3234 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3235 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3236 CKR_HOST_MEMORY, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED,
3237 CKR_OK, CKR_SAVED_STATE_INVALID, CKR_SESSION_CLOSED,
3238 CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

3239 Example:

```
3240 CK_SESSION_HANDLE hSession;  
3241 CK_MECHANISM digestMechanism;  
3242 CK_BYTE_PTR pState;  
3243 CK_ULONG ulStateLen;  
3244 CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};  
3245 CK_BYTE data2[] = {0x02, 0x04, 0x08};  
3246 CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};  
3247 CK_BYTE pDigest[20];  
3248 CK_ULONG ulDigestLen;  
3249 CK_RV rv;  
3250  
3251 .  
3252 .  
3253 /* Initialize hash operation */  
3254 rv = C_DigestInit(hSession, &digestMechanism);  
3255 assert(rv == CKR_OK);  
3256  
3257 /* Start hashing */  
3258 rv = C_DigestUpdate(hSession, data1, sizeof(data1));  
3259 assert(rv == CKR_OK);  
3260  
3261 /* Find out how big the state might be */  
3262 rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);  
3263 assert(rv == CKR_OK);  
3264  
3265 /* Allocate some memory and then get the state */  
3266 pState = (CK_BYTE_PTR) malloc(ulStateLen);  
3267 rv = C_GetOperationState(hSession, pState, &ulStateLen);  
3268
```

```

3269  /* Continue hashing */
3270  rv = C_DigestUpdate(hSession, data2, sizeof(data2));
3271  assert(rv == CKR_OK);
3272
3273  /* Restore state. No key handles needed */
3274  rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);
3275  assert(rv == CKR_OK);
3276
3277  /* Continue hashing from where we saved state */
3278  rv = C_DigestUpdate(hSession, data3, sizeof(data3));
3279  assert(rv == CKR_OK);
3280
3281  /* Conclude hashing operation */
3282  ulDigestLen = sizeof(pDigest);
3283  rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
3284  if (rv == CKR_OK) {
3285      /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
3286      .
3287      .
3288  }

```

3289 5.6.8 C_Login

```

3290 CK_DECLARE_FUNCTION(CK_RV, C_Login)(
3291     CK_SESSION_HANDLE hSession,
3292     CK_USER_TYPE userType,
3293     CK_UTF8CHAR_PTR pPin,
3294     CK_ULONG ulPinLen
3295 );

```

3296 **C_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to
3297 the user's PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid
3298 UTF8 character, but the token may impose subset restrictions.

3299 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
3300 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
3301 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of C_Login depends on the
3302 context in which it is called. Improper use of this user type will result in a return value
3303 CKR_OPERATION_NOT_INITIALIZED..

3304 If the token has a "protected authentication path", as indicated by the
3305 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
3306 that there is some way for a user to be authenticated to the token without having to send a PIN through
3307 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3308 on the slot device. Or the user might not even use a PIN—authentication could be achieved by some
3309 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
3310 parameter to **C_Login** should be NULL_PTR. When **C_Login** returns, whatever authentication method
3311 supported by the token will have been performed; a return value of CKR_OK means that the user was
3312 successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
3313 denied access.

3314 If there are any active cryptographic or object finding operations in an application's session, and then
3315 **C_Login** is successfully executed by that application, it may or may not be the case that those operations
3316 are still active. Therefore, before logging in, any active operations should be finished.

3317 If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the
3318 SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error code
3319 CKR_SESSION_READ_ONLY_EXISTS.

3320 C_Login may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
3321 CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do
3322 cryptographic operation on this key. See further Section 4.9.

3323 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3324 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3325 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3326 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT,
3327 CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3328 CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN,
3329 CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED,
3330 CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

3331 Example: see **C_Logout**.

3332 5.6.9 C_LoginUser

```
3333 CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) (  
3334     CK_SESSION_HANDLE hSession,  
3335     CK_USER_TYPE userType,  
3336     CK_UTF8CHAR_PTR pPin,  
3337     CK_ULONG ulPinLen,  
3338     CK_UTF8CHAR_PTR pUsername,  
3339     CK_ULONG ulUsernameLen  
3340 );
```

3341 **C_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin*
3342 points to the user's PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name,
3343 *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to
3344 contain any valid UTF8 character, but the token may impose subset restrictions.

3345 When the user type is either CKU_SO or CKU_USER, if the call succeeds, each of the application's
3346 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
3347 Functions" state. If the user type is CKU_CONTEXT_SPECIFIC, the behavior of **C_LoginUser** depends
3348 on the context in which it is called. Improper use of this user type will result in a return value
3349 CKR_OPERATION_NOT_INITIALIZED.

3350 If the token has a "protected authentication path", as indicated by the
3351 CKF_PROTECTED_AUTHENTICATION_PATH flag in its CK_TOKEN_INFO being set, then that means
3352 that there is some way for a user to be authenticated to the token without having to send a PIN through
3353 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3354 on the slot device. The user might not even use a PIN—authentication could be achieved by some
3355 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
3356 parameter to **C_LoginUser** should be NULL_PTR. When **C_LoginUser** returns, whatever authentication
3357 method supported by the token will have been performed; a return value of CKR_OK means that the user
3358 was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
3359 denied access.

3360 If there are any active cryptographic or object finding operations in an application's session, and then
3361 **C_LoginUser** is successfully executed by that application, it may or may not be the case that those
3362 operations are still active. Therefore, before logging in, any active operations should be finished.

3363 If the application calling **C_LoginUser** has a R/O session open with the token, then it will be unable to log
3364 the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error
3365 code CKR_SESSION_READ_ONLY_EXISTS.

C_LoginUser may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the CKA_ALWAYS_AUTHENTICATE attribute set to CK_TRUE exists, and the user needs to do cryptographic operation on this key. See further Section 4.9.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_PIN_INCORRECT, CKR_PIN_LOCKED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY_EXISTS, CKR_USER_ALREADY_LOGGED_IN, CKR_USER_ANOTHER_ALREADY_LOGGED_IN, CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

Example:

```
CK_SESSION_HANDLE hSession;
CK_UTF8CHAR userPin[] = {"MyPIN"};
CK_UTF8CHAR userName[] = {"MyUserName"};
CK_RV rv;

rv = C_LoginUser(hSession, CKU_USER, userPin, sizeof(userPin)-1, userName,
sizeof(userName)-1);
if (rv == CKR_OK) {
    .
    .
    rv = C_Logout(hSession);
    if (rv == CKR_OK) {
        .
        .
    }
}
```

5.6.10 C_Logout

```
CK_DECLARE_FUNCTION(CK_RV, C_Logout)(
    CK_SESSION_HANDLE hSession
);
```

C_Logout logs a user out from a token. *hSession* is the session's handle.

Depending on the current user type, if the call succeeds, each of the application's sessions will enter either the "R/W Public Session" state or the "R/O Public Session" state.

When **C_Logout** successfully executes, any of the application's handles to private objects become invalid (even if a user is later logged back into the token, those handles remain invalid). In addition, all private session objects from sessions belonging to the application are destroyed.

If there are any active cryptographic or object-finding operations in an application's session, and then **C_Logout** is successfully executed by that application, it may or may not be the case that those operations are still active. Therefore, before logging out, any active operations should be finished.

Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example:


```

3412 CK_SESSION_HANDLE hSession;
3413 CK_UTF8CHAR userPin[] = {"MyPIN"};
3414 CK_RV rv;
3415
3416 rv = C_Login(hSession, CKU_USER, userPin, sizeof(userPin)-1);
3417 if (rv == CKR_OK) {
3418     .
3419     .
3420     rv = C_Logout(hSession);
3421     if (rv == CKR_OK) {
3422         .
3423         .
3424     }
3425 }

```

5.7 Object management functions

Cryptoki provides the following functions for managing objects. Additional functions provided specifically for managing key objects are described in Section 5.18.

5.7.1 C_CreateObject

```

3430 CK_DECLARE_FUNCTION(CK_RV, C_CreateObject)(
3431     CK_SESSION_HANDLE hSession,
3432     CK_ATTRIBUTE_PTR pTemplate,
3433     CK_ULONG ulCount,
3434     CK_OBJECT_HANDLE_PTR phObject
3435 );

```

C_CreateObject creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives the new object's handle.

If a call to **C_CreateObject** cannot support the precise template supplied to it, it will fail and return without creating any object.

If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to CK_FALSE. If that key object is a secret or private key then the new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE.

Only session objects can be created during a read-only session. Only public objects can be created unless the normal user is logged in.

Whenever an object is created, a value for CKA_UNIQUE_ID is generated and assigned to the new object (See Section 4.4.1).

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

Example:

```

3458 CK_SESSION_HANDLE hSession;
3459 CK_OBJECT_HANDLE
3460     hData,
3461     hCertificate,
3462     hKey;
3463 CK_OBJECT_CLASS
3464     dataClass = CKO_DATA,
3465     certificateClass = CKO_CERTIFICATE,
3466     keyClass = CKO_PUBLIC_KEY;
3467 CK_KEY_TYPE keyType = CKK_RSA;
3468 CK_UTF8CHAR application[] = {"My Application"};
3469 CK_BYTE dataValue[] = {...};
3470 CK_BYTE subject[] = {...};
3471 CK_BYTE id[] = {...};
3472 CK_BYTE certificateValue[] = {...};
3473 CK_BYTE modulus[] = {...};
3474 CK_BYTE exponent[] = {...};
3475 CK_BBOOL true = CK_TRUE;
3476 CK_ATTRIBUTE dataTemplate[] = {
3477     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3478     {CKA_TOKEN, &true, sizeof(true)},
3479     {CKA_APPLICATION, application, sizeof(application)-1},
3480     {CKA_VALUE, dataValue, sizeof(dataValue)}
3481 };
3482 CK_ATTRIBUTE certificateTemplate[] = {
3483     {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
3484     {CKA_TOKEN, &true, sizeof(true)},
3485     {CKA_SUBJECT, subject, sizeof(subject)},
3486     {CKA_ID, id, sizeof(id)},
3487     {CKA_VALUE, certificateValue, sizeof(certificateValue)}
3488 };
3489 CK_ATTRIBUTE keyTemplate[] = {
3490     {CKA_CLASS, &keyClass, sizeof(keyClass)},
3491     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
3492     {CKA_WRAP, &true, sizeof(true)},
3493     {CKA_MODULUS, modulus, sizeof(modulus)},
3494     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
3495 };
3496 CK_RV rv;
3497
3498 .
3499 .
3500 /* Create a data object */

```



```

3501 rv = C_CreateObject(hSession, dataTemplate, 4, &hData);
3502 if (rv == CKR_OK) {
3503     .
3504     .
3505 }
3506
3507 /* Create a certificate object */
3508 rv = C_CreateObject(
3509     hSession, certificateTemplate, 5, &hCertificate);
3510 if (rv == CKR_OK) {
3511     .
3512     .
3513 }
3514
3515 /* Create an RSA public key object */
3516 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
3517 if (rv == CKR_OK) {
3518     .
3519     .
3520 }

```

3521 5.7.2 C_CopyObject

```

3522 CK_DECLARE_FUNCTION(CK_RV, C_CopyObject)(
3523     CK_SESSION_HANDLE hSession,
3524     CK_OBJECT_HANDLE hObject,
3525     CK_ATTRIBUTE_PTR pTemplate,
3526     CK_ULONG ulCount,
3527     CK_OBJECT_HANDLE_PTR phNewObject
3528 );

```

3529 **C_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session's handle;
3530 *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number
3531 of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of
3532 the object.

3533 The template may specify new values for any attributes of the object that can ordinarily be modified (e.g.,
3534 in the course of copying a secret key, a key's **CKA_EXTRACTABLE** attribute may be changed from
3535 CK_TRUE to CK_FALSE, but not the other way around. If this change is made, the new key's
3536 **CKA_NEVER_EXTRACTABLE** attribute will have the value CK_FALSE. Similarly, the template may
3537 specify that the new key's **CKA_SENSITIVE** attribute be CK_TRUE; the new key will have the same
3538 value for its **CKA_ALWAYS_SENSITIVE** attribute as the original key). It may also specify new values of
3539 the **CKA_TOKEN** and **CKA_PRIVATE** attributes (e.g., to copy a session object to a token object). If the
3540 template specifies a value of an attribute which is incompatible with other existing attributes of the object,
3541 the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

3542 If a call to **C_CopyObject** cannot support the precise template supplied to it, it will fail and return without
3543 creating any object. If the object indicated by *hObject* has its CKA_COPYABLE attribute set to
3544 CK_FALSE, C_CopyObject will return CKR_ACTION_PROHIBITED.

3545 Whenever an object is copied, a new value for CKA_UNIQUE_ID is generated and assigned to the new
3546 object (See Section 4.4.1).

3547 Only session objects can be created during a read-only session. Only public objects can be created
3548 unless the normal user is logged in.

3549 Return values: , CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
3550 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
3551 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3552 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3553 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3554 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
3555 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCONSISTENT,
3556 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

3557 Example:

```
3558 CK_SESSION_HANDLE hSession;  
3559 CK_OBJECT_HANDLE hKey, hNewKey;  
3560 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
3561 CK_KEY_TYPE keyType = CKK_DES;  
3562 CK_BYTE id[] = {...};  
3563 CK_BYTE keyValue[] = {...};  
3564 CK_BBOOL false = CK_FALSE;  
3565 CK_BBOOL true = CK_TRUE;  
3566 CK_ATTRIBUTE keyTemplate[] = {  
3567     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
3568     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
3569     {CKA_TOKEN, &>false, sizeof(false)},  
3570     {CKA_ID, id, sizeof(id)},  
3571     {CKA_VALUE, keyValue, sizeof(keyValue)}  
3572 };  
3573 CK_ATTRIBUTE copyTemplate[] = {  
3574     {CKA_TOKEN, &true, sizeof(true)}  
3575 };  
3576 CK_RV rv;  
3577  
3578 .  
3579 .  
3580 /* Create a DES secret key session object */  
3581 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);  
3582 if (rv == CKR_OK) {  
3583     /* Create a copy which is a token object */  
3584     rv = C_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);  
3585     .  
3586     .  
3587 }
```

3588 5.7.3 C_DestroyObject

```
3589 CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject) (  
3590     CK_SESSION_HANDLE hSession,
```

```

3591     CK_OBJECT_HANDLE hObject
3592 );

```

3593 **C_DestroyObject** destroys an object. *hSession* is the session's handle; and *hObject* is the object's
3594 handle.

3595 Only session objects can be destroyed during a read-only session. Only public objects can be destroyed
3596 unless the normal user is logged in.

3597 Certain objects may not be destroyed. Calling **C_DestroyObject** on such objects will result in the
3598 CKR_ACTION_PROHIBITED error code. An application can consult the object's CKA_DESTROYABLE
3599 attribute to determine if an object may be destroyed or not.

3600 Return values: CKR_ACTION_PROHIBITED, CKR_CRYPTOKI_NOT_INITIALIZED,
3601 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3602 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
3603 CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3604 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
3605 CKR_TOKEN_WRITE_PROTECTED.

3606 Example: see **C_GetObjectSize**.

3607 5.7.4 C_GetObjectSize

```

3608 CK_DECLARE_FUNCTION(CK_RV, C_GetObjectSize)(
3609     CK_SESSION_HANDLE hSession,
3610     CK_OBJECT_HANDLE hObject,
3611     CK_ULONG_PTR pulSize
3612 );

```

3613 **C_GetObjectSize** gets the size of an object in bytes. *hSession* is the session's handle; *hObject* is the
3614 object's handle; *pulSize* points to the location that receives the size in bytes of the object.

3615 Cryptoki does not specify what the precise meaning of an object's size is. Intuitively, it is some measure
3616 of how much token memory the object takes up. If an application deletes (say) a private object of size *S*,
3617 it might be reasonable to assume that the *ulFreePrivateMemory* field of the token's **CK_TOKEN_INFO**
3618 structure increases by approximately *S*.

3619 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3620 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3621 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
3622 CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3623 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3624 Example:

```

3625 CK_SESSION_HANDLE hSession;
3626 CK_OBJECT_HANDLE hObject;
3627 CK_OBJECT_CLASS dataClass = CKO_DATA;
3628 CK_UTF8CHAR application[] = {"My Application"};
3629 CK_BYTE value[] = {...};
3630 CK_BBOOL true = CK_TRUE;
3631 CK_ATTRIBUTE template[] = {
3632     {CKA_CLASS, &dataClass, sizeof(dataClass)},
3633     {CKA_TOKEN, &true, sizeof(true)},
3634     {CKA_APPLICATION, application, sizeof(application)-1},
3635     {CKA_VALUE, value, sizeof(value)}
3636 };
3637 CK_ULONG ulSize;

```

```

3638 CK_RV rv;
3639
3640 .
3641 .
3642 rv = C_CreateObject(hSession, template, 4, &hObject);
3643 if (rv == CKR_OK) {
3644     rv = C_GetObjectSize(hSession, hObject, &ulSize);
3645     if (rv != CKR_INFORMATION_SENSITIVE) {
3646         .
3647         .
3648     }
3649
3650     rv = C_DestroyObject(hSession, hObject);
3651     .
3652     .
3653 }

```

3654 5.7.5 C_GetAttributeValue

```

3655 CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue) (
3656     CK_SESSION_HANDLE hSession,
3657     CK_OBJECT_HANDLE hObject,
3658     CK_ATTRIBUTE_PTR pTemplate,
3659     CK_ULONG ulCount
3660 );

```

3661 **C_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session's
3662 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute
3663 values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the
3664 template.

3665 For each (*type*, *pValue*, *ulValueLen*) triple in the template, **C_GetAttributeValue** performs the following
3666 algorithm:

- 3667 1. If the specified attribute (i.e., the attribute specified by the type field) for the object cannot be revealed
3668 because the object is sensitive or unextractable, then the *ulValueLen* field in that triple is modified to
3669 hold the value CK_UNAVAILABLE_INFORMATION.
- 3670 2. Otherwise, if the specified value for the object is invalid (the object does not possess such an
3671 attribute), then the *ulValueLen* field in that triple is modified to hold the value
3672 CK_UNAVAILABLE_INFORMATION.
- 3673 3. Otherwise, if the *pValue* field has the value NULL_PTR, then the *ulValueLen* field is modified to hold
3674 the exact length of the specified attribute for the object.
- 3675 4. Otherwise, if the length specified in *ulValueLen* is large enough to hold the value of the specified
3676 attribute for the object, then that attribute is copied into the buffer located at *pValue*, and the
3677 *ulValueLen* field is modified to hold the exact length of the attribute.
- 3678 5. Otherwise, the *ulValueLen* field is modified to hold the value CK_UNAVAILABLE_INFORMATION.

3679 If case 1 applies to any of the requested attributes, then the call should return the value
3680 CKR_ATTRIBUTE_SENSITIVE. If case 2 applies to any of the requested attributes, then the call should
3681 return the value CKR_ATTRIBUTE_TYPE_INVALID. If case 5 applies to any of the requested attributes,
3682 then the call should return the value CKR_BUFFER_TOO_SMALL. As usual, if more than one of these
3683 error codes is applicable, Cryptoki may return any of them. Only if none of them applies to any of the
3684 requested attributes will CKR_OK be returned.

3685 In the special case of an attribute whose value is an array of attributes, for example
 3686 CKA_WRAP_TEMPLATE, where it is passed in with pValue not NULL, the length specified in ulValueLen
 3687 MUST be large enough to hold all attributes in the array. If the pValue of elements within the array is
 3688 NULL_PTR then the ulValueLen of elements within the array will be set to the required length. If the
 3689 pValue of elements within the array is not NULL_PTR, then the ulValueLen element of attributes within
 3690 the array MUST reflect the space that the corresponding pValue points to, and pValue is filled in if there is
 3691 sufficient room. Therefore it is important to initialize the contents of a buffer before calling
 3692 C_GetAttributeValue to get such an array value. Note that the type element of attributes within the array
 3693 MUST be ignored on input and MUST be set on output. If any ulValueLen within the array isn't large
 3694 enough, it will be set to CK_UNAVAILABLE_INFORMATION and the function will return
 3695 CKR_BUFFER_TOO_SMALL, as it does if an attribute in the pTemplate argument has ulValueLen too
 3696 small. Note that any attribute whose value is an array of attributes is identifiable by virtue of the attribute
 3697 type having the CKF_ARRAY_ATTRIBUTE bit set.

3698 Note that the error codes CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, and
 3699 CKR_BUFFER_TOO_SMALL do not denote true errors for **C_GetAttributeValue**. If a call to
 3700 **C_GetAttributeValue** returns any of these three values, then the call MUST nonetheless have processed
 3701 every attribute in the template supplied to **C_GetAttributeValue**. Each attribute in the template whose
 3702 value *can be* returned by the call to **C_GetAttributeValue** *will be* returned by the call to
 3703 **C_GetAttributeValue**.

3704 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE,
 3705 CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL,
 3706 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 3707 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 3708 CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_SESSION_CLOSED,
 3709 CKR_SESSION_HANDLE_INVALID.

3710 Example:

```
3711 CK_SESSION_HANDLE hSession;
3712 CK_OBJECT_HANDLE hObject;
3713 CK_BYTE_PTR pModulus, pExponent;
3714 CK_ATTRIBUTE template[] = {
3715     {CKA_MODULUS, NULL_PTR, 0},
3716     {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
3717 };
3718 CK_RV rv;
3719
3720 .
3721 .
3722 rv = C_GetAttributeValue(hSession, hObject, template, 2);
3723 if (rv == CKR_OK) {
3724     pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);
3725     template[0].pValue = pModulus;
3726     /* template[0].ulValueLen was set by C_GetAttributeValue */
3727
3728     pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);
3729     template[1].pValue = pExponent;
3730     /* template[1].ulValueLen was set by C_GetAttributeValue */
3731
3732     rv = C_GetAttributeValue(hSession, hObject, template, 2);
```

```

3733     if (rv == CKR_OK) {
3734         .
3735         .
3736     }
3737     free(pModulus);
3738     free(pExponent);
3739 }

```

3740 5.7.6 C_SetAttributeValue

```

3741 CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue) (
3742     CK_SESSION_HANDLE hSession,
3743     CK_OBJECT_HANDLE hObject,
3744     CK_ATTRIBUTE_PTR pTemplate,
3745     CK_ULONG ulCount
3746 );

```

3747 **C_SetAttributeValue** modifies the value of one or more attributes of an object. *hSession* is the session's
3748 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute
3749 values are to be modified and their new values; *ulCount* is the number of attributes in the template.

3750 Certain objects may not be modified. Calling C_SetAttributeValue on such objects will result in the
3751 CKR_ACTION_PROHIBITED error code. An application can consult the object's CKA_MODIFIABLE
3752 attribute to determine if an object may be modified or not.

3753 Only session objects can be modified during a read-only session.

3754 The template may specify new values for any attributes of the object that can be modified. If the template
3755 specifies a value of an attribute which is incompatible with other existing attributes of the object, the call
3756 fails with the return code CKR_TEMPLATE_INCONSISTENT.

3757 Not all attributes can be modified; see Section 4.1.2 for more details.

3758 Return values: CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
3759 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
3760 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3761 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3762 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
3763 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
3764 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
3765 CKR_USER_NOT_LOGGED_IN.

3766 Example:

```

3767 CK_SESSION_HANDLE hSession;
3768 CK_OBJECT_HANDLE hObject;
3769 CK_UTF8CHAR label[] = {"New label"};
3770 CK_ATTRIBUTE template[] = {
3771     {CKA_LABEL, label, sizeof(label)-1}
3772 };
3773 CK_RV rv;
3774
3775 .
3776 .
3777 rv = C_SetAttributeValue(hSession, hObject, template, 1);
3778 if (rv == CKR_OK) {

```

3779
3780
3781

```
.  
.}  

```

3782 5.7.7 C_FindObjectsInit

```
3783 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit) (  
3784     CK_SESSION_HANDLE hSession,  
3785     CK_ATTRIBUTE_PTR pTemplate,  
3786     CK_ULONG ulCount  
3787 );
```

3788 **C_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is
3789 the session's handle; *pTemplate* points to a search template that specifies the attribute values to match;
3790 *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-
3791 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

3792 After calling **C_FindObjectsInit**, the application may call **C_FindObjects** one or more times to obtain
3793 handles for objects matching the template, and then eventually call **C_FindObjectsFinal** to finish the
3794 active search operation. At most one search operation may be active at a given time in a given session.

3795 The object search operation will only find objects that the session can view. For example, an object
3796 search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the
3797 search template specifies that the search is for private objects).

3798 If a search operation is active, and objects are created or destroyed which fit the search template for the
3799 active search operation, then those objects may or may not be found by the search operation. Note that
3800 this means that, under these circumstances, the search operation may return invalid object handles.

3801 Even though **C_FindObjectsInit** can return the values CKR_ATTRIBUTE_TYPE_INVALID and
3802 CKR_ATTRIBUTE_VALUE_INVALID, it is not required to. For example, if it is given a search template
3803 with nonexistent attributes in it, it can return CKR_ATTRIBUTE_TYPE_INVALID, or it can initialize a
3804 search operation which will match no objects and return CKR_OK.

3805 If the CKA_UNIQUE_ID attribute is present in the search template, either zero or one objects will be
3806 found, since at most one object can have any particular CKA_UNIQUE_ID value.

3807 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID,
3808 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
3809 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
3810 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
3811 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3812 Example: see **C_FindObjectsFinal**.

3813 5.7.8 C_FindObjects

```
3814 CK_DECLARE_FUNCTION(CK_RV, C_FindObjects) (  
3815     CK_SESSION_HANDLE hSession,  
3816     CK_OBJECT_HANDLE_PTR phObject,  
3817     CK_ULONG ulMaxObjectCount,  
3818     CK_ULONG_PTR pulObjectCount  
3819 );
```

3820 **C_FindObjects** continues a search for token and session objects that match a template, obtaining
3821 additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives
3822 the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles
3823 to be returned; *pulObjectCount* points to the location that receives the actual number of object handles
3824 returned.

3825 If there are no more objects matching the template, then the location that *pulObjectCount* points to
3826 receives the value 0.

3827 The search MUST have been initialized with **C_FindObjectsInit**.

3828 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 3829 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 3830 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
 3831 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3832 Example: see **C_FindObjectsFinal**.

3833 5.7.9 C_FindObjectsFinal

```
3834 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsFinal) (
3835     CK_SESSION_HANDLE hSession
3836 );
```

3837 **C_FindObjectsFinal** terminates a search for token and session objects. *hSession* is the session's
 3838 handle.

3839 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 3840 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 3841 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 3842 CKR_SESSION_HANDLE_INVALID.

3843 Example:

```
3844 CK_SESSION_HANDLE hSession;
3845 CK_OBJECT_HANDLE hObject;
3846 CK_ULONG ulObjectCount;
3847 CK_RV rv;
3848
3849 .
3850 .
3851 rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
3852 assert(rv == CKR_OK);
3853 while (1) {
3854     rv = C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
3855     if (rv != CKR_OK || ulObjectCount == 0)
3856         break;
3857     .
3858     .
3859 }
3860
3861 rv = C_FindObjectsFinal(hSession);
3862 assert(rv == CKR_OK);
```

3863 5.8 Encryption functions

3864 Cryptoki provides the following functions for encrypting data:

3865 5.8.1 C_EncryptInit

```
3866 CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit) (
3867     CK_SESSION_HANDLE hSession,
3868     CK_MECHANISM_PTR pMechanism,
```



```
3869     CK_OBJECT_HANDLE hKey
3870 );
```

3871 **C_EncryptInit** initializes an encryption operation. *hSession* is the session's handle; *pMechanism* points
3872 to the encryption mechanism; *hKey* is the handle of the encryption key.

3873 The **CKA_ENCRYPT** attribute of the encryption key, which indicates whether the key supports
3874 encryption, MUST be CK_TRUE.

3875 After calling **C_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or
3876 call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts.
3877 The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** to
3878 *actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the
3879 application MUST call **C_EncryptInit** again.

3880 **C_EncryptInit** can be called with *pMechanism* set to NULL_PTR to terminate an active encryption
3881 operation. If an active operation operations cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
3882 must be returned.

3883 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3884 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
3885 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED,
3886 CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT,
3887 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
3888 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
3889 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
3890 CKR_OPERATION_CANCEL_FAILED.

3891 Example: see **C_EncryptFinal**.

3892 5.8.2 C_Encrypt

```
3893 CK_DECLARE_FUNCTION(CK_RV, C_Encrypt)(
3894     CK_SESSION_HANDLE hSession,
3895     CK_BYTE_PTR pData,
3896     CK_ULONG ulDataLen,
3897     CK_BYTE_PTR pEncryptedData,
3898     CK_ULONG_PTR pulEncryptedDataLen
3899 );
```

3900 **C_Encrypt** encrypts single-part data. *hSession* is the session's handle; *pData* points to the data;
3901 *ulDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the
3902 encrypted data; *pulEncryptedDataLen* points to the location that holds the length in bytes of the encrypted
3903 data.

3904 **C_Encrypt** uses the convention described in Section 5.2 on producing output.

3905 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always
3906 terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3907 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
3908 ciphertext.

3909 **C_Encrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_EncryptInit**
3910 without intervening **C_EncryptUpdate** calls.

3911 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
3912 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
3913 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
3914 **C_Encrypt** will fail with return code CKR_DATA_LEN_RANGE.

3915 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to
3916 the same location.

3917 For most mechanisms, **C_Encrypt** is equivalent to a sequence of **C_EncryptUpdate** operations followed
3918 by **C_EncryptFinal**.

3919 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3920 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
3921 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
3922 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3923 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
3924 CKR_SESSION_HANDLE_INVALID.

3925 Example: see **C_EncryptFinal** for an example of similar functions.

3926 5.8.3 C_EncryptUpdate

```
3927 CK_DECLARE_FUNCTION(CK_RV, C_EncryptUpdate) (  
3928     CK_SESSION_HANDLE hSession,  
3929     CK_BYTE_PTR pPart,  
3930     CK_ULONG ulPartLen,  
3931     CK_BYTE_PTR pEncryptedPart,  
3932     CK_ULONG_PTR pulEncryptedPartLen  
3933 );
```

3934 **C_EncryptUpdate** continues a multiple-part encryption operation, processing another data part.
3935 *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part;
3936 *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points
3937 to the location that holds the length in bytes of the encrypted data part.

3938 **C_EncryptUpdate** uses the convention described in Section 5.2 on producing output.

3939 The encryption operation MUST have been initialized with **C_EncryptInit**. This function may be called
3940 any number of times in succession. A call to **C_EncryptUpdate** which results in an error other than
3941 CKR_BUFFER_TOO_SMALL terminates the current encryption operation.

3942 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pPart* and *pEncryptedPart* point to
3943 the same location.

3944 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3945 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
3946 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
3947 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3948 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3949 Example: see **C_EncryptFinal**.

3950 5.8.4 C_EncryptFinal

```
3951 CK_DECLARE_FUNCTION(CK_RV, C_EncryptFinal) (  
3952     CK_SESSION_HANDLE hSession,  
3953     CK_BYTE_PTR pLastEncryptedPart,  
3954     CK_ULONG_PTR pulLastEncryptedPartLen  
3955 );
```

3956 **C_EncryptFinal** finishes a multiple-part encryption operation. *hSession* is the session's handle;
3957 *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any;
3958 *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

3959 **C_EncryptFinal** uses the convention described in Section 5.2 on producing output.

3960 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_EncryptFinal**
3961 always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
3962 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
3963 ciphertext.

3964 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
3965 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
3966 are not satisfied, then **C_EncryptFinal** will fail with return code CKR_DATA_LEN_RANGE.

3967 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
3968 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
3969 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
3970 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
3971 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3972 Example:

```
3973 #define PLAINTEXT_BUF_SZ 200
3974 #define CIPHERTEXT_BUF_SZ 256
3975
3976 CK_ULONG firstPieceLen, secondPieceLen;
3977 CK_SESSION_HANDLE hSession;
3978 CK_OBJECT_HANDLE hKey;
3979 CK_BYTE iv[8];
3980 CK_MECHANISM mechanism = {
3981     CKM_DES_CBC_PAD, iv, sizeof(iv)
3982 };
3983 CK_BYTE data[PLAINTEXT_BUF_SZ];
3984 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
3985 CK_ULONG ulEncryptedData1Len;
3986 CK_ULONG ulEncryptedData2Len;
3987 CK_ULONG ulEncryptedData3Len;
3988 CK_RV rv;
3989
3990 .
3991 .
3992 firstPieceLen = 90;
3993 secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
3994 rv = C_EncryptInit(hSession, &mechanism, hKey);
3995 if (rv == CKR_OK) {
3996     /* Encrypt first piece */
3997     ulEncryptedData1Len = sizeof(encryptedData);
3998     rv = C_EncryptUpdate(
3999         hSession,
4000         &data[0], firstPieceLen,
4001         &encryptedData[0], &ulEncryptedData1Len);
4002     if (rv != CKR_OK) {
4003         .
4004         .
4005     }
4006
4007     /* Encrypt second piece */
4008     ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
4009     rv = C_EncryptUpdate(
4010         hSession,
```

```

4011     &data[firstPieceLen], secondPieceLen,
4012     &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
4013 if (rv != CKR_OK) {
4014     .
4015     .
4016 }
4017
4018 /* Get last little encrypted bit */
4019 ulEncryptedData3Len =
4020     sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
4021 rv = C_EncryptFinal(
4022     hSession,
4023     &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
4024     &ulEncryptedData3Len);
4025 if (rv != CKR_OK) {
4026     .
4027     .
4028 }
4029 }

```

5.9 Message-based encryption functions

Message-based encryption refers to the process of encrypting multiple messages using the same encryption mechanism and encryption key. The encryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

Cryptoki provides the following functions for message-based encryption:

5.9.1 C_MessageEncryptInit

```

CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptInit)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);

```

C_MessageEncryptInit prepares a session for one or more encryption operations that use the same encryption mechanism and encryption key. *hSession* is the session's handle; *pMechanism* points to the encryption mechanism; *hKey* is the handle of the encryption key.

The CKA_ENCRYPT attribute of the encryption key, which indicates whether the key supports encryption, MUST be CK_TRUE.

After calling **C_MessageEncryptInit**, the application can either call **C_EncryptMessage** to encrypt a message in a single part, or call **C_EncryptMessageBegin**, followed by **C_EncryptMessageNext** one or more times, to encrypt a message in multiple parts. This may be repeated several times. The message-based encryption process is active until the application calls **C_MessageEncryptFinal** to finish the message-based encryption process.

C_MessageEncryptInit can be called with *pMechanism* set to NULL_PTR to terminate a message-based encryption process. If a multi-part message encryption operation is active, it will also be terminated. If an active operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.


```

4105     CK_SESSION_HANDLE hSession,
4106     CK_VOID_PTR pParameter,
4107     CK_ULONG ulParameterLen,
4108     CK_BYTE_PTR pAssociatedData,
4109     CK_ULONG ulAssociatedDataLen
4110 );

```

4111 **C_EncryptMessageBegin** begins a multiple-part message encryption operation. *hSession* is the
4112 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
4113 message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
4114 for an AEAD mechanism.

4115 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
4116 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
4117 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
4118 generator will be output to the *pParameter* buffer.

4119 If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be
4120 set to (NULL, 0).

4121 After calling **C_EncryptMessageBegin**, the application should call **C_EncryptMessageNext** one or
4122 more times to encrypt the message in multiple parts. The message encryption operation is active until the
4123 application uses a call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE to actually
4124 obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the
4125 application MUST call **C_EncryptMessage** or **C_EncryptMessageBegin** again.

4126 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4127 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4128 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
4129 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4130 CKR_USER_NOT_LOGGED_IN.

4131 5.9.4 C_EncryptMessageNext

```

4132 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (
4133     CK_SESSION_HANDLE hSession,
4134     CK_VOID_PTR pParameter,
4135     CK_ULONG ulParameterLen,
4136     CK_BYTE_PTR pPlaintextPart,
4137     CK_ULONG ulPlaintextPartLen,
4138     CK_BYTE_PTR pCiphertextPart,
4139     CK_ULONG_PTR pulCiphertextPartLen,
4140     CK_FLAGS flags
4141 );

```

4142 **C_EncryptMessageNext** continues a multiple-part message encryption operation, processing another
4143 message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
4144 mechanism-specific parameters for the message encryption operation; *pPlaintextPart* points to the
4145 plaintext message part; *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart*
4146 points to the location that receives the encrypted message part; *pulCiphertextPartLen* points to the
4147 location that holds the length in bytes of the encrypted message part; flags is set to 0 if there is more
4148 plaintext data to follow, or set to CKF_END_OF_MESSAGE if this is the last plaintext part.

4149 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
4150 passed to **C_EncryptMessageNext**, *pParameter* may be either an input or an output parameter. For
4151 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
4152 generator will be output to the *pParameter* buffer.

4153 **C_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

4154 The message encryption operation MUST have been started with **C_EncryptMessageBegin**. This
4155 function may be called any number of times in succession. A call to **C_EncryptMessageNext** with flags=0
4156 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message

4157 encryption operation. A call to **C_EncryptMessageNext** with flags=CKF_END_OF_MESSAGE always
4158 terminates the active message encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4159 successful call (i.e., one which returns **CKR_OK**) to determine the length of the buffer needed to hold the
4160 ciphertext.

4161 Although the last **C_EncryptMessageNext** call ends the encryption of a message, it does not finish the
4162 message-based encryption process. Additional **C_EncryptMessage** or **C_EncryptMessageBegin** and
4163 **C_EncryptMessageNext** calls may be made on the session.

4164 The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart*
4165 point to the same location.

4166 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
4167 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
4168 are not satisfied when the final message part is supplied (i.e., with flags=CKF_END_OF_MESSAGE),
4169 then **C_EncryptMessageNext** will fail with return code CKR_DATA_LEN_RANGE.

4170 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4171 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4172 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4173 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4174 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4175 5.9.5 C_MessageEncryptFinal

```
4176 CK_DECLARE_FUNCTION(CK_RV, C_MessageEncryptFinal)(  
4177     CK_SESSION_HANDLE hSession  
4178 );
```

4179 **C_MessageEncryptFinal** finishes a message-based encryption process. *hSession* is the session's
4180 handle.

4181 The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**.

4182 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4183 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4184 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4185 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4186 CKR_SESSION_HANDLE_INVALID.

4187 Example:

```
4188 #define PLAINTEXT_BUF_SZ 200  
4189 #define AUTH_BUF_SZ 100  
4190 #define CIPHERTEXT_BUF_SZ 256  
4191  
4192 CK_SESSION_HANDLE hSession;  
4193 CK_OBJECT_HANDLE hKey;  
4194 CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };  
4195 CK_BYTE tag[16];  
4196 CK_GCM_MESSAGE_PARAMS gcmParams = {  
4197     iv,  
4198     sizeof(iv) * 8,  
4199     0,  
4200     CKG_NO_GENERATE,  
4201     tag,  
4202     sizeof(tag) * 8
```

```

4203 };
4204 CK_MECHANISM mechanism = {
4205     CKM_AES_GCM, &gcmParams, sizeof(gcmParams)
4206 };
4207 CK_BYTE data[2][PLAINTEXT_BUF_SZ];
4208 CK_BYTE auth[2][AUTH_BUF_SZ];
4209 CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];
4210 CK_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;
4211 CK_ULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;
4212
4213 /* error handling is omitted for better readability */
4214 .
4215 .
4216 C_MessageEncryptInit(hSession, &mechanism, hKey);
4217 /* encrypt message en bloc with given IV */
4218 ulEncryptedDataLen = sizeof(encryptedData[0]);
4219 C_EncryptMessage(hSession,
4220     &gcmParams, sizeof(gcmParams),
4221     &auth[0][0], sizeof(auth[0]),
4222     &data[0][0], sizeof(data[0]),
4223     &encryptedData[0][0], &ulEncryptedDataLen);
4224 /* iv and tag are set now for message */
4225
4226 /* encrypt message in two steps with generated IV */
4227 gcmParams.ivGenerator = CKG_GENERATE;
4228 C_EncryptMessageBegin(hSession,
4229     &gcmParams, sizeof(gcmParams),
4230     &auth[1][0], sizeof(auth[1])
4231 );
4232 /* encrypt first piece */
4233 ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
4234 C_EncryptMessageNext(hSession,
4235     &gcmParams, sizeof(gcmParams),
4236     &data[1][0], firstPieceLen,
4237     &encryptedData[1][0], &ulFirstEncryptedDataLen,
4238     0
4239 );
4240 /* encrypt second piece */
4241 ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
4242 C_EncryptMessageNext(hSession,
4243     &gcmParams, sizeof(gcmParams),
4244     &data[1][firstPieceLen], sizeof(data[1]) - firstPieceLen,
4245     &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,

```



```

4246     CKF_END_OF_MESSAGE
4247 );
4248 /* tag is set now for message */
4249
4250 /* finalize */
4251 C_MessageEncryptFinal(hSession);

```

5.10 Decryption functions

Cryptoki provides the following functions for decrypting data:

5.10.1 C_DecryptInit

```

4255 CK_DECLARE_FUNCTION(CK_RV, C_DecryptInit) (
4256     CK_SESSION_HANDLE hSession,
4257     CK_MECHANISM_PTR pMechanism,
4258     CK_OBJECT_HANDLE hKey
4259 );

```

C_DecryptInit initializes a decryption operation. *hSession* is the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the decryption key.

The **CKA_DECRYPT** attribute of the decryption key, which indicates whether the key supports decryption, **MUST** be **CK_TRUE**.

After calling **C_DecryptInit**, the application can either call **C_Decrypt** to decrypt data in a single part; or call **C_DecryptUpdate** zero or more times, followed by **C_DecryptFinal**, to decrypt data in multiple parts. The decryption operation is active until the application uses a call to **C_Decrypt** or **C_DecryptFinal** to *actually obtain* the final piece of plaintext. To process additional data (in single or multiple parts), the application **MUST** call **C_DecryptInit** again.

C_DecryptInit can be called with *pMechanism* set to **NULL_PTR** to terminate an active decryption operation. If an active operation cannot be cancelled, **CKR_OPERATION_CANCEL_FAILED** must be returned.

Return values: **CKR_ARGUMENTS_BAD**, **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_KEY_FUNCTION_NOT_PERMITTED**, **CKR_KEY_HANDLE_INVALID**, **CKR_KEY_SIZE_RANGE**, **CKR_KEY_TYPE_INCONSISTENT**, **CKR_MECHANISM_INVALID**, **CKR_MECHANISM_PARAM_INVALID**, **CKR_OK**, **CKR_OPERATION_ACTIVE**, **CKR_PIN_EXPIRED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**, **CKR_USER_NOT_LOGGED_IN**, **CKR_OPERATION_CANCEL_FAILED**.

Example: see **C_DecryptFinal**.

5.10.2 C_Decrypt

```

4282 CK_DECLARE_FUNCTION(CK_RV, C_Decrypt) (
4283     CK_SESSION_HANDLE hSession,
4284     CK_BYTE_PTR pEncryptedData,
4285     CK_ULONG ulEncryptedDataLen,
4286     CK_BYTE_PTR pData,
4287     CK_ULONG_PTR pulDataLen
4288 );

```

C_Decrypt decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData* points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the

4291 location that receives the recovered data; *pulDataLen* points to the location that holds the length of the
 4292 recovered data.

4293 **C_Decrypt** uses the convention described in Section 5.2 on producing output.

4294 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_Decrypt** always
 4295 terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
 4296 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
 4297 plaintext.

4298 **C_Decrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_DecryptInit**
 4299 without intervening **C_DecryptUpdate** calls.

4300 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to
 4301 the same location.

4302 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
 4303 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4304 For most mechanisms, **C_Decrypt** is equivalent to a sequence of **C_DecryptUpdate** operations followed
 4305 by **C_DecryptFinal**.

4306 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4307 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4308 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
 4309 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 4310 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
 4311 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4312 Example: see **C_DecryptFinal** for an example of similar functions.

4313 5.10.3 C_DecryptUpdate

```

4314 CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) (
4315     CK_SESSION_HANDLE hSession,
4316     CK_BYTE_PTR pEncryptedPart,
4317     CK_ULONG ulEncryptedPartLen,
4318     CK_BYTE_PTR pPart,
4319     CK_ULONG_PTR pulPartLen
4320 );
  
```

4321 **C_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data
 4322 part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part;
 4323 *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the
 4324 recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

4325 **C_DecryptUpdate** uses the convention described in Section 5.2 on producing output.

4326 The decryption operation MUST have been initialized with **C_DecryptInit**. This function may be called
 4327 any number of times in succession. A call to **C_DecryptUpdate** which results in an error other than
 4328 CKR_BUFFER_TOO_SMALL terminates the current decryption operation.

4329 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to
 4330 the same location.

4331 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4332 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4333 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
 4334 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 4335 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
 4336 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4337 Example: See **C_DecryptFinal**.

5.10.4 C_DecryptFinal

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal)(
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pLastPart,
    CK_ULONG_PTR pullLastPartLen
);
```

C_DecryptFinal finishes a multiple-part decryption operation. *hSession* is the session's handle; *pLastPart* points to the location that receives the last recovered data part, if any; *pullLastPartLen* points to the location that holds the length of the last recovered data part.

C_DecryptFinal uses the convention described in Section 5.2 on producing output.

The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_DecryptFinal** always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the plaintext.

If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID, CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

Example:

```
#define CIPHERTEXT_BUF_SZ 256
#define PLAINTEXT_BUF_SZ 256

CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[8];
CK_MECHANISM mechanism = {
    CKM_DES_CBC_PAD, iv, sizeof(iv)
};

CK_BYTE data[PLAINTEXT_BUF_SZ];
CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
CK_ULONG ulData1Len, ulData2Len, ulData3Len;
CK_RV rv;

.
.

firstEncryptedPieceLen = 90;
secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ-firstEncryptedPieceLen;
rv = C_DecryptInit(hSession, &mechanism, hKey);
if (rv == CKR_OK) {
    /* Decrypt first piece */
    ulData1Len = sizeof(data);
```

```

4384     rv = C_DecryptUpdate(
4385         hSession,
4386         &encryptedData[0], firstEncryptedPieceLen,
4387         &data[0], &ulData1Len);
4388     if (rv != CKR_OK) {
4389         .
4390         .
4391     }
4392
4393     /* Decrypt second piece */
4394     ulData2Len = sizeof(data)-ulData1Len;
4395     rv = C_DecryptUpdate(
4396         hSession,
4397         &encryptedData[firstEncryptedPieceLen],
4398         secondEncryptedPieceLen,
4399         &data[ulData1Len], &ulData2Len);
4400     if (rv != CKR_OK) {
4401         .
4402         .
4403     }
4404
4405     /* Get last little decrypted bit */
4406     ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
4407     rv = C_DecryptFinal(
4408         hSession,
4409         &data[ulData1Len+ulData2Len], &ulData3Len);
4410     if (rv != CKR_OK) {
4411         .
4412         .
4413     }
4414 }

```

5.11 Message-based decryption functions

Message-based decryption refers to the process of decrypting multiple encrypted messages using the same decryption mechanism and decryption key. The decryption mechanism can be either an authenticated encryption with associated data (AEAD) algorithm or a pure encryption algorithm.

Cryptoki provides the following functions for message-based decryption.

5.11.1 C_MessageDecryptInit

```

4421 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit)(
4422     CK_SESSION_HANDLE hSession,
4423     CK_MECHANISM_PTR pMechanism,
4424     CK_OBJECT_HANDLE hKey
4425 );

```

4426 **C_MessageDecryptInit** initializes a message-based decryption process, preparing a session for one or
4427 more decryption operations that use the same decryption mechanism and decryption key. *hSession* is
4428 the session's handle; *pMechanism* points to the decryption mechanism; *hKey* is the handle of the
4429 decryption key.

4430 The CKA_DECRYPT attribute of the decryption key, which indicates whether the key supports decryption,
4431 MUST be CK_TRUE.

4432 After calling **C_MessageDecryptInit**, the application can either call **C_DecryptMessage** to decrypt an
4433 encrypted message in a single part; or call **C_DecryptMessageBegin**, followed by

4434 **C_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This
4435 may be repeated several times. The message-based decryption process is active until the application
4436 uses a call to **C_MessageDecryptFinal** to finish the message-based decryption process.

4437 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4438 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4439 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4440 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4441 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4442 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4443 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4444 CKR_OPERATION_CANCEL_FAILED.

4445 5.11.2 C_DecryptMessage

```
4446 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) (  
4447     CK_SESSION_HANDLE hSession,  
4448     CK_VOID_PTR pParameter,  
4449     CK_ULONG ulParameterLen,  
4450     CK_BYTE_PTR pAssociatedData,  
4451     CK_ULONG ulAssociatedDataLen,  
4452     CK_BYTE_PTR pCiphertext,  
4453     CK_ULONG ulCiphertextLen,  
4454     CK_BYTE_PTR pPlaintext,  
4455     CK_ULONG_PTR pulPlaintextLen  
4456 );
```

4457 **C_DecryptMessage** decrypts an encrypted message in a single part. *hSession* is the session's handle;
4458 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption
4459 operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD
4460 mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted
4461 message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to
4462 the location that holds the length of the recovered message.

4463 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
4464 **C_EncryptMessage**, *pParameter* is always an input parameter.

4465 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
4466 should be set to (NULL, 0).

4467 **C_DecryptMessage** uses the convention described in Section 5.2 on producing output.

4468 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**. A call
4469 to **C_DecryptMessage** begins and terminates a message decryption operation.

4470 **C_DecryptMessage** cannot be called in the middle of a multi-part message decryption operation.

4471 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to
4472 the same location.

4473 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4474 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

4475 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
4476 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned.

4477 For most mechanisms, **C_DecryptMessage** is equivalent to **C_DecryptMessageBegin** followed by a
4478 sequence of **C_DecryptMessageNext** operations.

4479 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4480 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4481 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4482 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
4483 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4484 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4485 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4486 CKR_OPERATION_CANCEL_FAILED.

4487 5.11.3 C_DecryptMessageBegin

```
4488 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) (  
4489     CK_SESSION_HANDLE hSession,  
4490     CK_VOID_PTR pParameter,  
4491     CK_ULONG ulParameterLen,  
4492     CK_BYTE_PTR pAssociatedData,  
4493     CK_ULONG ulAssociatedDataLen  
4494 );
```

4495 **C_DecryptMessageBegin** begins a multiple-part message decryption operation. *hSession* is the
4496 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
4497 message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
4498 for an AEAD mechanism.

4499 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
4500 **C_EncryptMessageBegin**, *pParameter* is always an input parameter.

4501 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
4502 should be set to (NULL, 0).

4503 After calling **C_DecryptMessageBegin**, the application should call **C_DecryptMessageNext** one or
4504 more times to decrypt the encrypted message in multiple parts. The message decryption operation is
4505 active until the application uses a call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE
4506 to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or
4507 multiple parts), the application MUST call **C_DecryptMessage** or **C_DecryptMessageBegin** again.

4508 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4509 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4510 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4511 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4512 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4513 5.11.4 C_DecryptMessageNext

```
4514 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) (  
4515     CK_SESSION_HANDLE hSession,  
4516     CK_VOID_PTR pParameter,  
4517     CK_ULONG ulParameterLen,  
4518     CK_BYTE_PTR pCiphertextPart,  
4519     CK_ULONG ulCiphertextPartLen,  
4520     CK_BYTE_PTR pPlaintextPart,  
4521     CK_ULONG_PTR pulPlaintextPartLen,  
4522     CK_FLAGS flags  
4523 );
```

4524 **C_DecryptMessageNext** continues a multiple-part message decryption operation, processing another
4525 encrypted message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
4526 mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the

4527 encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart*
4528 points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location
4529 that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to
4530 follow, or set to CKF_END_OF_MESSAGE if this is the last ciphertext part.

4531 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
4532 **C_EncryptMessageNext**, *pParameter* is always an input parameter.

4533 **C_DecryptMessageNext** uses the convention described in Section 5.2 on producing output.

4534 The message decryption operation MUST have been started with **C_DecryptMessageBegin**. This
4535 function may be called any number of times in succession. A call to **C_DecryptMessageNext** with
4536 flags=0 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message
4537 decryption operation. A call to **C_DecryptMessageNext** with flags=CKF_END_OF_MESSAGE always
4538 terminates the active message decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4539 successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the
4540 plaintext.

4541 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart*
4542 point to the same location.

4543 Although the last **C_DecryptMessageNext** call ends the decryption of a message, it does not finish the
4544 message-based decryption process. Additional **C_DecryptMessage** or **C_DecryptMessageBegin** and
4545 **C_DecryptMessageNext** calls may be made on the session.

4546 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
4547 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned by
4548 the last **C_DecryptMessageNext** call.

4549 If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or
4550 ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned by the last
4551 **C_DecryptMessageNext** call.

4552 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4553 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4554 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
4555 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_AEAD_DECRYPT_FAILED,
4556 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4557 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4558 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4559 5.11.5 C_MessageDecryptFinal

```
4560 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptFinal) (  
4561     CK_SESSION_HANDLE hSession  
4562 );
```

4563 **C_MessageDecryptFinal** finishes a message-based decryption process. *hSession* is the session's
4564 handle.

4565 The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**.

4566 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4567 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4568 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4569 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4570 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4571 5.12 Message digesting functions

4572 Cryptoki provides the following functions for digesting data:

5.12.1 C_DigestInit

```
CK_DECLARE_FUNCTION(CK_RV, C_DigestInit)(  
    CK_SESSION_HANDLE hSession,  
    CK_MECHANISM_PTR pMechanism  
);
```

C_DigestInit initializes a message-digesting operation. *hSession* is the session's handle; *pMechanism* points to the digesting mechanism.

After calling **C_DigestInit**, the application can either call **C_Digest** to digest data in a single part; or call **C_DigestUpdate** zero or more times, followed by **C_DigestFinal**, to digest data in multiple parts. The message-digesting operation is active until the application uses a call to **C_Digest** or **C_DigestFinal** to *actually obtain* the message digest. To process additional data (in single or multiple parts), the application **MUST** call **C_DigestInit** again.

C_DigestInit can be called with *pMechanism* set to `NULL_PTR` to terminate an active message-digesting operation. If an operation has been initialized and it cannot be cancelled, `CKR_OPERATION_CANCEL_FAILED` must be returned.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_MECHANISM_INVALID`, `CKR_MECHANISM_PARAM_INVALID`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_PIN_EXPIRED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_USER_NOT_LOGGED_IN`, `CKR_OPERATION_CANCEL_FAILED`.

Example: see **C_DigestFinal**.

5.12.2 C_Digest

```
CK_DECLARE_FUNCTION(CK_RV, C_Digest)(  
    CK_SESSION_HANDLE hSession,  
    CK_BYTE_PTR pData,  
    CK_ULONG ulDataLen,  
    CK_BYTE_PTR pDigest,  
    CK_ULONG_PTR pulDigestLen  
);
```

C_Digest digests data in a single part. *hSession* is the session's handle, *pData* points to the data; *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest; *pulDigestLen* points to the location that holds the length of the message digest.

C_Digest uses the convention described in Section 5.2 on producing output.

The digest operation **MUST** have been initialized with **C_DigestInit**. A call to **C_Digest** always terminates the active digest operation unless it returns `CKR_BUFFER_TOO_SMALL` or is a successful call (*i.e.*, one which returns `CKR_OK`) to determine the length of the buffer needed to hold the message digest.

C_Digest cannot be used to terminate a multi-part operation, and **MUST** be called after **C_DigestInit** without intervening **C_DigestUpdate** calls.

The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the same location.

C_Digest is equivalent to a sequence of **C_DigestUpdate** operations followed by **C_DigestFinal**.

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

4622 Example: see **C_DigestFinal** for an example of similar functions.

4623 5.12.3 C_DigestUpdate

```
4624 CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate) (  
4625     CK_SESSION_HANDLE hSession,  
4626     CK_BYTE_PTR pPart,  
4627     CK_ULONG ulPartLen  
4628 );
```

4629 **C_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part.
4630 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4631 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
4632 and **C_DigestKey** may be interspersed any number of times in any order. A call to **C_DigestUpdate**
4633 which results in an error terminates the current digest operation.

4634 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4635 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4636 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4637 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4638 CKR_SESSION_HANDLE_INVALID.

4639 Example: see **C_DigestFinal**.

4640 5.12.4 C_DigestKey

```
4641 CK_DECLARE_FUNCTION(CK_RV, C_DigestKey) (  
4642     CK_SESSION_HANDLE hSession,  
4643     CK_OBJECT_HANDLE hKey  
4644 );
```

4645 **C_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret
4646 key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

4647 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
4648 and **C_DigestUpdate** may be interspersed any number of times in any order.

4649 If the value of the supplied key cannot be digested purely for some reason related to its length,
4650 **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.

4651 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4652 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4653 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
4654 CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK,
4655 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4656 Example: see **C_DigestFinal**.

4657 5.12.5 C_DigestFinal

```
4658 CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal) (  
4659     CK_SESSION_HANDLE hSession,  
4660     CK_BYTE_PTR pDigest,  
4661     CK_ULONG_PTR pulDigestLen  
4662 );
```

4663 **C_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest.
4664 *hSession* is the session's handle; *pDigest* points to the location that receives the message digest;
4665 *pulDigestLen* points to the location that holds the length of the message digest.

4666 **C_DigestFinal** uses the convention described in Section 5.2 on producing output.

4667 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_DigestFinal** always
4668 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful

4669 call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the message
4670 digest.

4671 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4672 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4673 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4674 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4675 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4676 Example:

```
4677 CK_SESSION_HANDLE hSession;  
4678 CK_OBJECT_HANDLE hKey;  
4679 CK_MECHANISM mechanism = {  
4680     CKM_MD5, NULL_PTR, 0  
4681 };  
4682 CK_BYTE data[] = {...};  
4683 CK_BYTE digest[16];  
4684 CK_ULONG ulDigestLen;  
4685 CK_RV rv;  
4686  
4687 .  
4688 .  
4689 rv = C_DigestInit(hSession, &mechanism);  
4690 if (rv != CKR_OK) {  
4691     .  
4692     .  
4693 }  
4694  
4695 rv = C_DigestUpdate(hSession, data, sizeof(data));  
4696 if (rv != CKR_OK) {  
4697     .  
4698     .  
4699 }  
4700  
4701 rv = C_DigestKey(hSession, hKey);  
4702 if (rv != CKR_OK) {  
4703     .  
4704     .  
4705 }  
4706  
4707 ulDigestLen = sizeof(digest);  
4708 rv = C_DigestFinal(hSession, digest, &ulDigestLen);  
4709 .  
4710 .
```

5.13 Signing and MACing functions

Cryptoki provides the following functions for signing data (for the purposes of Cryptoki, these operations also encompass message authentication codes).

5.13.1 C_SignInit

```
CK_DECLARE_FUNCTION(CK_RV, C_SignInit) (
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_SignInit initializes a signature operation, where the signature is an appendix to the data. *hSession* is the session's handle; *pMechanism* points to the signature mechanism; *hKey* is the handle of the signature key.

The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with appendix, MUST be CK_TRUE.

After calling **C_SignInit**, the application can either call **C_Sign** to sign in a single part; or call **C_SignUpdate** one or more times, followed by **C_SignFinal**, to sign data in multiple parts. The signature operation is active until the application uses a call to **C_Sign** or **C_SignFinal** to *actually obtain* the signature. To process additional data (in single or multiple parts), the application MUST call **C_SignInit** again.

C_SignInit can be called with *pMechanism* set to NULL_PTR to terminate an active signature operation. If an operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

Example: see **C_SignFinal**.

5.13.2 C_Sign

```
CK_DECLARE_FUNCTION(CK_RV, C_Sign) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG_PTR pulSignatureLen
);
```

C_Sign signs data in a single part, where the signature is an appendix to the data. *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

C_Sign uses the convention described in Section 5.2 on producing output.

The signing operation MUST have been initialized with **C_SignInit**. A call to **C_Sign** always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

4758 **C_Sign** cannot be used to terminate a multi-part operation, and MUST be called after **C_SignInit** without
4759 intervening **C_SignUpdate** calls.

4760 For most mechanisms, **C_Sign** is equivalent to a sequence of **C_SignUpdate** operations followed by
4761 **C_SignFinal**.

4762 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4763 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4764 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4765 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4766 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4767 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4768 CKR_TOKEN_RESOURCE_EXCEEDED.

4769 Example: see **C_SignFinal** for an example of similar functions.

4770 5.13.3 C_SignUpdate

```
4771 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate) (  
4772     CK_SESSION_HANDLE hSession,  
4773     CK_BYTE_PTR pPart,  
4774     CK_ULONG ulPartLen  
4775 );
```

4776 **C_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is
4777 the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

4778 The signature operation MUST have been initialized with **C_SignInit**. This function may be called any
4779 number of times in succession. A call to **C_SignUpdate** which results in an error terminates the current
4780 signature operation.

4781 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4782 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4783 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
4784 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
4785 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4786 CKR_TOKEN_RESOURCE_EXCEEDED.

4787 Example: see **C_SignFinal**.

4788 5.13.4 C_SignFinal

```
4789 CK_DECLARE_FUNCTION(CK_RV, C_SignFinal) (  
4790     CK_SESSION_HANDLE hSession,  
4791     CK_BYTE_PTR pSignature,  
4792     CK_ULONG_PTR pulSignatureLen  
4793 );
```

4794 **C_SignFinal** finishes a multiple-part signature operation, returning the signature. *hSession* is the
4795 session's handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to
4796 the location that holds the length of the signature.

4797 **C_SignFinal** uses the convention described in Section 5.2 on producing output.

4798 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_SignFinal** always
4799 terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
4800 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

4801 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4802 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4803 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4804 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4805 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,

4806 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
4807 CKR_TOKEN_RESOURCE_EXCEEDED.

4808 Example:

```
4809 CK_SESSION_HANDLE hSession;  
4810 CK_OBJECT_HANDLE hKey;  
4811 CK_MECHANISM mechanism = {  
4812     CKM_DES_MAC, NULL_PTR, 0  
4813 };  
4814 CK_BYTE data[] = {...};  
4815 CK_BYTE mac[4];  
4816 CK_ULONG ulMacLen;  
4817 CK_RV rv;  
4818  
4819 .  
4820 .  
4821 rv = C_SignInit(hSession, &mechanism, hKey);  
4822 if (rv == CKR_OK) {  
4823     rv = C_SignUpdate(hSession, data, sizeof(data));  
4824     .  
4825     .  
4826     ulMacLen = sizeof(mac);  
4827     rv = C_SignFinal(hSession, mac, &ulMacLen);  
4828     .  
4829     .  
4830 }
```

4831 5.13.5 C_SignRecoverInit

```
4832 CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit)(  
4833     CK_SESSION_HANDLE hSession,  
4834     CK_MECHANISM_PTR pMechanism,  
4835     CK_OBJECT_HANDLE hKey  
4836 );
```

4837 **C_SignRecoverInit** initializes a signature operation, where the data can be recovered from the signature.
4838 *hSession* is the session's handle; *pMechanism* points to the structure that specifies the signature
4839 mechanism; *hKey* is the handle of the signature key.

4840 The **CKA_SIGN_RECOVER** attribute of the signature key, which indicates whether the key supports
4841 signatures where the data can be recovered from the signature, MUST be CK_TRUE.

4842 After calling **C_SignRecoverInit**, the application may call **C_SignRecover** to sign in a single part. The
4843 signature operation is active until the application uses a call to **C_SignRecover** to actually obtain the
4844 signature. To process additional data in a single part, the application MUST call **C_SignRecoverInit**
4845 again.

4846 **C_SignRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature
4847 with data recovery operation. If an active operation has been initialized and it cannot be cancelled,
4848 CKR_OPERATION_CANCEL_FAILED must be returned.

4849 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4850 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,

4851 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4852 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4853 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4854 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4855 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4856 CKR_OPERATION_CANCEL_FAILED.

4857 Example: see **C_SignRecover**.

4858 5.13.6 C_SignRecover

```
4859 CK_DECLARE_FUNCTION(CK_RV, C_SignRecover) (  
4860     CK_SESSION_HANDLE hSession,  
4861     CK_BYTE_PTR pData,  
4862     CK_ULONG ulDataLen,  
4863     CK_BYTE_PTR pSignature,  
4864     CK_ULONG_PTR pulSignatureLen  
4865 );
```

4866 **C_SignRecover** signs data in a single operation, where the data can be recovered from the signature.
4867 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
4868 *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that
4869 holds the length of the signature.

4870 **C_SignRecover** uses the convention described in Section 5.2 on producing output.

4871 The signing operation MUST have been initialized with **C_SignRecoverInit**. A call to **C_SignRecover**
4872 always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4873 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4874 signature.

4875 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4876 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4877 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4878 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4879 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
4880 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4881 CKR_TOKEN_RESOURCE_EXCEEDED.

4882 Example:

```
4883 CK_SESSION_HANDLE hSession;  
4884 CK_OBJECT_HANDLE hKey;  
4885 CK_MECHANISM mechanism = {  
4886     CKM_RSA_9796, NULL_PTR, 0  
4887 };  
4888 CK_BYTE data[] = {...};  
4889 CK_BYTE signature[128];  
4890 CK_ULONG ulSignatureLen;  
4891 CK_RV rv;  
4892  
4893 .  
4894 .  
4895 rv = C_SignRecoverInit(hSession, &mechanism, hKey);  
4896 if (rv == CKR_OK) {  
4897     ulSignatureLen = sizeof(signature);
```

```

4898     rv = C_SignRecover(
4899         hSession, data, sizeof(data), signature, &ulSignatureLen);
4900     if (rv == CKR_OK) {
4901         .
4902         .
4903     }
4904 }
4905

```

4906 5.14 Message-based signing and MACing functions

4907 Message-based signature refers to the process of signing multiple messages using the same signature
4908 mechanism and signature key.

4909 Cryptoki provides the following functions for for signing messages (for the purposes of Cryptoki, these
4910 operations also encompass message authentication codes).

4911 5.14.1 C_MessageSignInit

```

4912 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignInit)(
4913     CK_SESSION_HANDLE hSession,
4914     CK_MECHANISM_PTR pMechanism,
4915     CK_OBJECT_HANDLE hKey
4916 );

```

4917 **C_MessageSignInit** initializes a message-based signature process, preparing a session for one or more
4918 signature operations (where the signature is an appendix to the data) that use the same signature
4919 mechanism and signature key. *hSession* is the session's handle; *pMechanism* points to the signature
4920 mechanism; *hKey* is the handle of the signature key.

4921 The **CKA_SIGN** attribute of the signature key, which indicates whether the key supports signatures with
4922 appendix, MUST be CK_TRUE.

4923 After calling **C_MessageSignInit**, the application can either call **C_SignMessage** to sign a message in a
4924 single part; or call **C_SignMessageBegin**, followed by **C_SignMessageNext** one or more times, to sign
4925 a message in multiple parts. This may be repeated several times. The message-based signature process
4926 is active until the application calls **C_MessageSignFinal** to finish the message-based signature process.

4927 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4928 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4929 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4930 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
4931 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
4932 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
4933 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4934 5.14.2 C_SignMessage

```

4935 CK_DECLARE_FUNCTION(CK_RV, C_SignMessage)(
4936     CK_SESSION_HANDLE hSession,
4937     CK_VOID_PTR pParameter,
4938     CK_ULONG ulParameterLen,
4939     CK_BYTE_PTR pData,
4940     CK_ULONG ulDataLen,

```



```

4941     CK_BYTE_PTR pSignature,
4942     CK_ULONG_PTR pulSignatureLen
4943 );

```

4944 **C_SignMessage** signs a message in a single part, where the signature is an appendix to the message.
 4945 **C_MessageSignInit** must previously been called on the session. *hSession* is the session's handle;
 4946 *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature
 4947 operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the location
 4948 that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

4949 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
 4950 input or an output parameter.

4951 **C_SignMessage** uses the convention described in Section 5.2 on producing output.

4952 The message-based signing process MUST have been initialized with **C_MessageSignInit**. A call to
 4953 **C_SignMessage** begins and terminates a message signing operation unless it returns
 4954 CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a
 4955 successful call (i.e., one which returns CKR_OK).

4956 **C_SignMessage** cannot be called in the middle of a multi-part message signing operation.

4957 **C_SignMessage** does not finish the message-based signing process. Additional **C_SignMessage** or
 4958 **C_SignMessageBegin** and **C_SignMessageNext** calls may be made on the session.

4959 For most mechanisms, **C_SignMessage** is equivalent to **C_SignMessageBegin** followed by a sequence
 4960 of **C_SignMessageNext** operations.

4961 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4962 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
 4963 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4964 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4965 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
 4966 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
 4967 CKR_TOKEN_RESOURCE_EXCEEDED.

4968 5.14.3 C_SignMessageBegin

```

4969 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageBegin) (
4970     CK_SESSION_HANDLE hSession,
4971     CK_VOID_PTR pParameter,
4972     CK_ULONG ulParameterLen
4973 );

```

4974 **C_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an
 4975 appendix to the message. **C_MessageSignInit** must previously been called on the session. *hSession* is
 4976 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
 4977 the message signature operation.

4978 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
 4979 input or an output parameter.

4980 After calling **C_SignMessageBegin**, the application should call **C_SignMessageNext** one or more times
 4981 to sign the message in multiple parts. The message signature operation is active until the application
 4982 uses a call to **C_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature.
 4983 To process additional messages (in single or multiple parts), the application MUST call **C_SignMessage**
 4984 or **C_SignMessageBegin** again.

4985 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4986 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4987 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4988 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,

4989 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
4990 CKR_TOKEN_RESOURCE_EXCEEDED.

4991 5.14.4 C_SignMessageNext

```
4992 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) (  
4993     CK_SESSION_HANDLE hSession,  
4994     CK_VOID_PTR pParameter,  
4995     CK_ULONG ulParameterLen,  
4996     CK_BYTE_PTR pDataPart,  
4997     CK_ULONG ulDataPartLen,  
4998     CK_BYTE_PTR pSignature,  
4999     CK_ULONG_PTR pulSignatureLen  
5000 );
```

5001 **C_SignMessageNext** continues a multiple-part message signature operation, processing another data
5002 part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the
5003 session's handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any
5004 mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the
5005 data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the
5006 location that holds the length of the signature.

5007 The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NULL
5008 value (to receive the signature length) if this is the last data part.

5009 **C_SignMessageNext** uses the convention described in Section 5.2 on producing output.

5010 The message signing operation MUST have been started with **C_SignMessageBegin**. This function may
5011 be called any number of times in succession. A call to **C_SignMessageNext** with a NULL
5012 *pulSignatureLen* which results in an error terminates the current message signature operation. A call to
5013 **C_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing
5014 operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to
5015 hold the signature, or is a successful call (i.e., one which returns CKR_OK).

5016 Although the last **C_SignMessageNext** call ends the signing of a message, it does not finish the
5017 message-based signing process. Additional **C_SignMessage** or **C_SignMessageBegin** and
5018 **C_SignMessageNext** calls may be made on the session.

5019 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5020 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
5021 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5022 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
5023 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5024 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
5025 CKR_TOKEN_RESOURCE_EXCEEDED.

5026 5.14.5 C_MessageSignFinal

```
5027 CK_DECLARE_FUNCTION(CK_RV, C_MessageSignFinal) (  
5028     CK_SESSION_HANDLE hSession  
5029 );
```

5030 **C_MessageSignFinal** finishes a message-based signing process. *hSession* is the session's handle.

5031 The message-based signing process MUST have been initialized with **C_MessageSignInit**.

5032 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5033 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5034 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,

5035 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
5036 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
5037 CKR_TOKEN_RESOURCE_EXCEEDED.

5038 5.15 Functions for verifying signatures and MACs

5039 Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki,
5040 these operations also encompass message authentication codes):

5041 5.15.1 C_VerifyInit

```
5042 CK_DECLARE_FUNCTION(CK_RV, C_VerifyInit) (  
5043     CK_SESSION_HANDLE hSession,  
5044     CK_MECHANISM_PTR pMechanism,  
5045     CK_OBJECT_HANDLE hKey  
5046 );
```

5047 **C_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession*
5048 is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;
5049 *hKey* is the handle of the verification key.

5050 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
5051 where the signature is an appendix to the data, MUST be CK_TRUE.

5052 After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single
5053 part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data
5054 in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**.
5055 To process additional data (in single or multiple parts), the application MUST call **C_VerifyInit** again.

5056 **C_VerifyInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
5057 operation. If an active operation has been initialized and it cannot be cancelled,
5058 CKR_OPERATION_CANCEL_FAILED must be returned.

5059 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5060 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5061 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5062 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5063 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5064 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5065 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5066 CKR_OPERATION_CANCEL_FAILED.

5067 Example: see **C_VerifyFinal**.

5068 5.15.2 C_Verify

```
5069 CK_DECLARE_FUNCTION(CK_RV, C_Verify) (  
5070     CK_SESSION_HANDLE hSession,  
5071     CK_BYTE_PTR pData,  
5072     CK_ULONG ulDataLen,  
5073     CK_BYTE_PTR pSignature,  
5074     CK_ULONG ulSignatureLen  
5075 );
```

5076 **C_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data.
5077 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
5078 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5079 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_Verify** always
5080 terminates the active verification operation.

5081 A successful call to **C_Verify** should return either the value CKR_OK (indicating that the supplied
5082 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the

5083 signature can be seen to be invalid purely on the basis of its length, then
5084 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active signing operation
5085 is terminated.

5086 **C_Verify** cannot be used to terminate a multi-part operation, and MUST be called after **C_VerifyInit**
5087 without intervening **C_VerifyUpdate** calls.

5088 For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by
5089 **C_VerifyFinal**.

5090 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
5091 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5092 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5093 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5094 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5095 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5096 Example: see **C_VerifyFinal** for an example of similar functions.

5097 5.15.3 C_VerifyUpdate

```
5098 CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate)(  
5099     CK_SESSION_HANDLE hSession,  
5100     CK_BYTE_PTR pPart,  
5101     CK_ULONG ulPartLen  
5102 );
```

5103 **C_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession*
5104 is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

5105 The verification operation MUST have been initialized with **C_VerifyInit**. This function may be called any
5106 number of times in succession. A call to **C_VerifyUpdate** which results in an error terminates the current
5107 verification operation.

5108 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5109 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5110 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5111 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5112 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5113 CKR_TOKEN_RESOURCE_EXCEEDED.

5114 Example: see **C_VerifyFinal**.

5115 5.15.4 C_VerifyFinal

```
5116 CK_DECLARE_FUNCTION(CK_RV, C_VerifyFinal)(  
5117     CK_SESSION_HANDLE hSession,  
5118     CK_BYTE_PTR pSignature,  
5119     CK_ULONG ulSignatureLen  
5120 );
```

5121 **C_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the
5122 session's handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5123 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_VerifyFinal** always
5124 terminates the active verification operation.

5125 A successful call to **C_VerifyFinal** should return either the value CKR_OK (indicating that the supplied
5126 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
5127 signature can be seen to be invalid purely on the basis of its length, then
5128 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active verifying
5129 operation is terminated.

5130 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5131 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,

5132 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5133 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5134 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5135 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5136 Example:

```
5137 CK_SESSION_HANDLE hSession;  
5138 CK_OBJECT_HANDLE hKey;  
5139 CK_MECHANISM mechanism = {  
5140     CKM_DES_MAC, NULL_PTR, 0  
5141 };  
5142 CK_BYTE data[] = {...};  
5143 CK_BYTE mac[4];  
5144 CK_RV rv;  
5145  
5146 .  
5147 .  
5148 rv = C_VerifyInit(hSession, &mechanism, hKey);  
5149 if (rv == CKR_OK) {  
5150     rv = C_VerifyUpdate(hSession, data, sizeof(data));  
5151     .  
5152     .  
5153     rv = C_VerifyFinal(hSession, mac, sizeof(mac));  
5154     .  
5155     .  
5156 }
```

5157 5.15.5 C_VerifyRecoverInit

```
5158 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecoverInit)(  
5159     CK_SESSION_HANDLE hSession,  
5160     CK_MECHANISM_PTR pMechanism,  
5161     CK_OBJECT_HANDLE hKey  
5162 );
```

5163 **C_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the
5164 signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the
5165 verification mechanism; *hKey* is the handle of the verification key.

5166 The **CKA_VERIFY_RECOVER** attribute of the verification key, which indicates whether the key supports
5167 verification where the data is recovered from the signature, MUST be CK_TRUE.

5168 After calling **C_VerifyRecoverInit**, the application may call **C_VerifyRecover** to verify a signature on
5169 data in a single part. The verification operation is active until the application uses a call to
5170 **C_VerifyRecover** to actually obtain the recovered message.

5171 **C_VerifyRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
5172 with data recovery operation. If an active operations has been initialized and it cannot be cancelled,
5173 CKR_OPERATION_CANCEL_FAILED must be returned.

5174 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5175 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5176 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5177 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,

5178 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5179 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5180 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5181 CKR_OPERATION_CANCEL_FAILED.

5182 Example: see **C_VerifyRecover**.

5183 5.15.6 C_VerifyRecover

```
5184 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecover) (  
5185     CK_SESSION_HANDLE hSession,  
5186     CK_BYTE_PTR pSignature,  
5187     CK_ULONG ulSignatureLen,  
5188     CK_BYTE_PTR pData,  
5189     CK_ULONG_PTR pulDataLen  
5190 );
```

5191 **C_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the
5192 signature. *hSession* is the session's handle; *pSignature* points to the signature; *ulSignatureLen* is the
5193 length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen*
5194 points to the location that holds the length of the recovered data.

5195 **C_VerifyRecover** uses the convention described in Section 5.2 on producing output.

5196 The verification operation MUST have been initialized with **C_VerifyRecoverInit**. A call to
5197 **C_VerifyRecover** always terminates the active verification operation unless it returns
5198 CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the
5199 length of the buffer needed to hold the recovered data.

5200 A successful call to **C_VerifyRecover** should return either the value CKR_OK (indicating that the
5201 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
5202 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
5203 CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID
5204 and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code
5205 CKR_BUFFER_TOO_SMALL, *i.e.*, if **C_VerifyRecover** is supplied with an invalid signature, it will never
5206 return CKR_BUFFER_TOO_SMALL.

5207 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5208 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
5209 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5210 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5211 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED,
5212 CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID,
5213 CKR_TOKEN_RESOURCE_EXCEEDED.

5214 Example:

```
5215 CK_SESSION_HANDLE hSession;  
5216 CK_OBJECT_HANDLE hKey;  
5217 CK_MECHANISM mechanism = {  
5218     CKM_RSA_9796, NULL_PTR, 0  
5219 };  
5220 CK_BYTE data[] = {...};  
5221 CK_ULONG ulDataLen;  
5222 CK_BYTE signature[128];  
5223 CK_RV rv;  
5224  
5225 .
```

```

5226 .
5227 rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
5228 if (rv == CKR_OK) {
5229     ulDataLen = sizeof(data);
5230     rv = C_VerifyRecover(
5231         hSession, signature, sizeof(signature), data, &ulDataLen);
5232     .
5233     .
5234 }

```

5235 5.16 Message-based functions for verifying signatures and MACs

5236 Message-based verification refers to the process of verifying signatures on multiple messages using the
5237 same verification mechanism and verification key.

5238 Cryptoki provides the following functions for verifying signatures on messages (for the purposes of
5239 Cryptoki, these operations also encompass message authentication codes).

5240 5.16.1 C_MessageVerifyInit

```

5241 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyInit) (
5242     CK_SESSION_HANDLE hSession,
5243     CK_MECHANISM_PTR pMechanism,
5244     CK_OBJECT_HANDLE hKey
5245 );

```

5246 **C_MessageVerifyInit** initializes a message-based verification process, preparing a session for one or
5247 more verification operations (where the signature is an appendix to the data) that use the same
5248 verification mechanism and verification key. *hSession* is the session's handle; *pMechanism* points to the
5249 structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

5250 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
5251 where the signature is an appendix to the data, MUST be CK_TRUE.

5252 After calling **C_MessageVerifyInit**, the application can either call **C_VerifyMessage** to verify a signature
5253 on a message in a single part; or call **C_VerifyMessageBegin**, followed by **C_VerifyMessageNext** one
5254 or more times, to verify a signature on a message in multiple parts. This may be repeated several times.
5255 The message-based verification process is active until the application calls **C_MessageVerifyFinal** to
5256 finish the message-based verification process.

5257 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5258 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5259 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5260 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5261 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5262 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5263 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5264 5.16.2 C_VerifyMessage

```

5265 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessage) (
5266     CK_SESSION_HANDLE hSession,
5267     CK_VOID_PTR pParameter,
5268     CK_ULONG ulParameterLen,
5269     CK_BYTE_PTR pData,

```



```

5270     CK_ULONG ulDataLen,
5271     CK_BYTE_PTR pSignature,
5272     CK_ULONG ulSignatureLen
5273 );

```

5274 **C_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an
5275 appendix to the data. **C_MessageVerifyInit** must previously been called on the session. *hSession* is the
5276 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5277 message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature*
5278 points to the signature; *ulSignatureLen* is the length of the signature.

5279 Unlike the *pParameter* parameter of **C_SignMessage**, *pParameter* is always an input parameter.

5280 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**. A call to
5281 **C_VerifyMessage** starts and terminates a message verification operation.

5282 A successful call to **C_VerifyMessage** should return either the value CKR_OK (indicating that the
5283 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
5284 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
5285 CKR_SIGNATURE_LEN_RANGE should be returned.

5286 **C_VerifyMessage** does not finish the message-based verification process. Additional **C_VerifyMessage**
5287 or **C_VerifyMessageBegin** and **C_VerifyMessageNext** calls may be made on the session.

5288 For most mechanisms, **C_VerifyMessage** is equivalent to **C_VerifyMessageBegin** followed by a
5289 sequence of **C_VerifyMessageNext** operations.

5290 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
5291 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5292 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5293 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5294 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
5295 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5296 5.16.3 C_VerifyMessageBegin

```

5297 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin) (
5298     CK_SESSION_HANDLE hSession,
5299     CK_VOID_PTR pParameter,
5300     CK_ULONG ulParameterLen
5301 );

```

5302 **C_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an
5303 appendix to the message. **C_MessageVerifyInit** must previously been called on the session. *hSession* is
5304 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
5305 the message verification operation.

5306 Unlike the *pParameter* parameter of **C_SignMessageBegin**, *pParameter* is always an input parameter.

5307 After calling **C_VerifyMessageBegin**, the application should call **C_VerifyMessageNext** one or more
5308 times to verify a signature on a message in multiple parts. The message verification operation is active
5309 until the application calls **C_VerifyMessageNext** with a non-NULL *pSignature*. To process additional
5310 messages (in single or multiple parts), the application MUST call **C_VerifyMessage** or
5311 **C_VerifyMessageBegin** again.

5312 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5313 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5314 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5315 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
5316 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.16.4 C_VerifyMessageNext

```
CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pDataPart,
    CK_ULONG ulDataPartLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG ulSignatureLen
);
```

C_VerifyMessageNext continues a multiple-part message verification operation, processing another data part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the session's handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message verification operation, *pPart* points to the data part; *ulPartLen* is the length of the data part; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value (pointing to the signature to verify) if this is the last data part.

The message verification operation MUST have been started with **C_VerifyMessageBegin**. This function may be called any number of times in succession. A call to **C_VerifyMessageNext** with a NULL *pSignature* which results in an error terminates the current message verification operation. A call to **C_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification operation.

A successful call to **C_VerifyMessageNext** with a non-NULL *pSignature* should return either the value CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active message verifying operation is terminated.

Although the last **C_VerifyMessageNext** call ends the verification of a message, it does not finish the message-based verification process. Additional **C_VerifyMessage** or **C_VerifyMessageBegin** and **C_VerifyMessageNext** calls may be made on the session.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID, CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

5.16.5 C_MessageVerifyFinal

```
CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyFinal) (
    CK_SESSION_HANDLE hSession
);
```

C_MessageVerifyFinal finishes a message-based verification process. *hSession* is the session's handle.

The message-based verification process MUST have been initialized with **C_MessageVerifyInit**.

Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,

5363 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5364 CKR_TOKEN_RESOURCE_EXCEEDED.

5365 5.17 Dual-function cryptographic functions

5366 Cryptoki provides the following functions to perform two cryptographic operations “simultaneously” within
5367 a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and
5368 from a token.

5369 5.17.1 C_DigestEncryptUpdate

```
5370 CK_DECLARE_FUNCTION(CK_RV, C_DigestEncryptUpdate) (  
5371     CK_SESSION_HANDLE hSession,  
5372     CK_BYTE_PTR pPart,  
5373     CK_ULONG ulPartLen,  
5374     CK_BYTE_PTR pEncryptedPart,  
5375     CK_ULONG_PTR pulEncryptedPartLen  
5376 );
```

5377 **C_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another
5378 data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the
5379 data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part;
5380 *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5381 **C_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5382 **C_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
5383 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5384 encrypted part output), then no plaintext is passed to the active digest operation.

5385 Digest and encryption operations **MUST** both be active (they **MUST** have been initialized with
5386 **C_DigestInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5387 succession, and may be interspersed with **C_DigestUpdate**, **C_DigestKey**, and **C_EncryptUpdate** calls
5388 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5389 **C_DigestKey**, however).

5390 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5391 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
5392 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5393 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
5394 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5395 Example:

```
5396 #define BUF_SZ 512  
5397  
5398 CK_SESSION_HANDLE hSession;  
5399 CK_OBJECT_HANDLE hKey;  
5400 CK_BYTE iv[8];  
5401 CK_MECHANISM digestMechanism = {  
5402     CKM_MD5, NULL_PTR, 0  
5403 };  
5404 CK_MECHANISM encryptionMechanism = {  
5405     CKM_DES_ECB, iv, sizeof(iv)  
5406 };  
5407 CK_BYTE encryptedData[BUF_SZ];  
5408 CK_ULONG ulEncryptedDataLen;
```

```

5409 CK_BYTE digest[16];
5410 CK_ULONG ulDigestLen;
5411 CK_BYTE data[(2*BUF_SZ)+8];
5412 CK_RV rv;
5413 int i;
5414
5415 .
5416 .
5417 memset(iv, 0, sizeof(iv));
5418 memset(data, 'A', ((2*BUF_SZ)+5));
5419 rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
5420 if (rv != CKR_OK) {
5421     .
5422     .
5423 }
5424 rv = C_DigestInit(hSession, &digestMechanism);
5425 if (rv != CKR_OK) {
5426     .
5427     .
5428 }
5429
5430 ulEncryptedDataLen = sizeof(encryptedData);
5431 rv = C_DigestEncryptUpdate(
5432     hSession,
5433     &data[0], BUF_SZ,
5434     encryptedData, &ulEncryptedDataLen);
5435 .
5436 .
5437 ulEncryptedDataLen = sizeof(encryptedData);
5438 rv = C_DigestEncryptUpdate(
5439     hSession,
5440     &data[BUF_SZ], BUF_SZ,
5441     encryptedData, &ulEncryptedDataLen);
5442 .
5443 .
5444
5445 /*
5446  * The last portion of the buffer needs to be
5447  * handled with separate calls to deal with
5448  * padding issues in ECB mode
5449  */
5450
5451 /* First, complete the digest on the buffer */

```

```

5452 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
5453 .
5454 .
5455 ulDigestLen = sizeof(digest);
5456 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5457 .
5458 .
5459
5460 /* Then, pad last part with 3 0x00 bytes, and complete encryption */
5461 for(i=0;i<3;i++)
5462     data[((BUF_SZ*2)+5)+i] = 0x00;
5463
5464 /* Now, get second-to-last piece of ciphertext */
5465 ulEncryptedDataLen = sizeof(encryptedData);
5466 rv = C_EncryptUpdate(
5467     hSession,
5468     &data[BUF_SZ*2], 8,
5469     encryptedData, &ulEncryptedDataLen);
5470 .
5471 .
5472
5473 /* Get last piece of ciphertext (should have length 0, here) */
5474 ulEncryptedDataLen = sizeof(encryptedData);
5475 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5476 .
5477 .

```

5478 5.17.2 C_DecryptDigestUpdate

```

5479 CK_DECLARE_FUNCTION(CK_RV, C_DecryptDigestUpdate) (
5480     CK_SESSION_HANDLE hSession,
5481     CK_BYTE_PTR pEncryptedPart,
5482     CK_ULONG ulEncryptedPartLen,
5483     CK_BYTE_PTR pPart,
5484     CK_ULONG_PTR pulPartLen
5485 );

```

5486 **C_DecryptDigestUpdate** continues a multiple-part combined decryption and digest operation,
5487 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
5488 data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that
5489 receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered
5490 data part.

5491 **C_DecryptDigestUpdate** uses the convention described in Section 5.2 on producing output. If a
5492 **C_DecryptDigestUpdate** call does not produce decrypted output (because an error occurs, or because
5493 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire decrypted part
5494 output), then no plaintext is passed to the active digest operation.

5495 Decryption and digesting operations **MUST** both be active (they **MUST** have been initialized with
5496 **C_DecryptInit** and **C_DigestInit**, respectively). This function may be called any number of times in

5497 succession, and may be interspersed with **C_DecryptUpdate**, **C_DigestUpdate**, and **C_DigestKey** calls
5498 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
5499 **C_DigestKey**, however).

5500 Use of **C_DecryptDigestUpdate** involves a pipelining issue that does not arise when using
5501 **C_DigestEncryptUpdate**, the “inverse function” of **C_DecryptDigestUpdate**. This is because when
5502 **C_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting
5503 operation and the active encryption operation; however, when **C_DecryptDigestUpdate** is called, the
5504 input passed to the active digesting operation is the *output* of the active decryption operation. This issue
5505 comes up only when the mechanism used for decryption performs padding.

5506 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
5507 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
5508 ciphertext and digest the original plaintext thereby obtained.

5509 After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES
5510 blocks) into **C_DecryptDigestUpdate**. **C_DecryptDigestUpdate** returns exactly 16 bytes of plaintext,
5511 since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of
5512 ciphertext held any padding. These 16 bytes of plaintext are passed into the active digesting operation.

5513 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's
5514 no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
5515 decryption and digesting operations are linked *only* through the **C_DecryptDigestUpdate** call, these 2
5516 bytes of plaintext are *not* passed on to be digested.

5517 A call to **C_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the*
5518 *plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C_DigestFinal** is called,
5519 the last 2 bytes of plaintext get passed into the active digesting operation via a **C_DigestUpdate** call.

5520 Because of this, it is critical that when an application uses a padded decryption mechanism with
5521 **C_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting
5522 operation. *Extreme caution is warranted when using a padded decryption mechanism with*
5523 **C_DecryptDigestUpdate**.

5524 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5525 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5526 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
5527 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5528 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
5529 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5530 Example:

```
5531 #define BUF_SZ 512
5532
5533 CK_SESSION_HANDLE hSession;
5534 CK_OBJECT_HANDLE hKey;
5535 CK_BYTE iv[8];
5536 CK_MECHANISM decryptionMechanism = {
5537     CKM_DES_ECB, iv, sizeof(iv)
5538 };
5539 CK_MECHANISM digestMechanism = {
5540     CKM_MD5, NULL_PTR, 0
5541 };
5542 CK_BYTE encryptedData[(2*BUF_SZ)+8];
5543 CK_BYTE digest[16];
5544 CK_ULONG ulDigestLen;
5545 CK_BYTE data[BUF_SZ];
```

```

5546 CK_ULONG ulDataLen, ulLastUpdateSize;
5547 CK_RV rv;
5548
5549 .
5550 .
5551 memset(iv, 0, sizeof(iv));
5552 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5553 rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
5554 if (rv != CKR_OK) {
5555     .
5556     .
5557 }
5558 rv = C_DigestInit(hSession, &digestMechanism);
5559 if (rv != CKR_OK){
5560     .
5561     .
5562 }
5563
5564 ulDataLen = sizeof(data);
5565 rv = C_DecryptDigestUpdate(
5566     hSession,
5567     &encryptedData[0], BUF_SZ,
5568     data, &ulDataLen);
5569 .
5570 .
5571 ulDataLen = sizeof(data);
5572 rv = C_DecryptDigestUpdate(
5573     hSession,
5574     &encryptedData[BUF_SZ], BUF_SZ,
5575     data, &ulDataLen);
5576 .
5577 .
5578
5579 /*
5580  * The last portion of the buffer needs to be handled with
5581  * separate calls to deal with padding issues in ECB mode
5582  */
5583
5584 /* First, complete the decryption of the buffer */
5585 ulLastUpdateSize = sizeof(data);
5586 rv = C_DecryptUpdate(
5587     hSession,
5588     &encryptedData[BUF_SZ*2], 8,

```

```

5589     data, &ulLastUpdateSize);
5590     .
5591     .
5592     /* Get last piece of plaintext (should have length 0, here) */
5593     ulDataLen = sizeof(data)-ulLastUpdateSize;
5594     rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5595     if (rv != CKR_OK) {
5596         .
5597         .
5598     }
5599
5600     /* Digest last bit of plaintext */
5601     rv = C_DigestUpdate(hSession, data, 5);
5602     if (rv != CKR_OK) {
5603         .
5604         .
5605     }
5606     ulDigestLen = sizeof(digest);
5607     rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5608     if (rv != CKR_OK) {
5609         .
5610         .
5611     }

```

5612 5.17.3 C_SignEncryptUpdate

```

5613 CK_DECLARE_FUNCTION(CK_RV, C_SignEncryptUpdate) (
5614     CK_SESSION_HANDLE hSession,
5615     CK_BYTE_PTR pPart,
5616     CK_ULONG ulPartLen,
5617     CK_BYTE_PTR pEncryptedPart,
5618     CK_ULONG_PTR pulEncryptedPartLen
5619 );

```

5620 **C_SignEncryptUpdate** continues a multiple-part combined signature and encryption operation,
5621 processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is
5622 the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted
5623 data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

5624 **C_SignEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
5625 **C_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
5626 *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire
5627 encrypted part output), then no plaintext is passed to the active signing operation.

5628 Signature and encryption operations **MUST** both be active (they **MUST** have been initialized with
5629 **C_SignInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
5630 succession, and may be interspersed with **C_SignUpdate** and **C_EncryptUpdate** calls.

5631 Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`,
5632 `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`,
5633 `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_FUNCTION_CANCELED`,
5634 `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`,

5635 CKR_OPERATION_NOT_INITIALIZED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5636 CKR_USER_NOT_LOGGED_IN.

5637 Example:

```
5638 #define BUF_SZ 512
5639
5640 CK_SESSION_HANDLE hSession;
5641 CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
5642 CK_BYTE iv[8];
5643 CK_MECHANISM signMechanism = {
5644     CKM_DES_MAC, NULL_PTR, 0
5645 };
5646 CK_MECHANISM encryptionMechanism = {
5647     CKM_DES_ECB, iv, sizeof(iv)
5648 };
5649 CK_BYTE encryptedData[BUF_SZ];
5650 CK_ULONG ulEncryptedDataLen;
5651 CK_BYTE MAC[4];
5652 CK_ULONG ulMacLen;
5653 CK_BYTE data[(2*BUF_SZ)+8];
5654 CK_RV rv;
5655 int i;
5656
5657 .
5658 .
5659 memset(iv, 0, sizeof(iv));
5660 memset(data, 'A', ((2*BUF_SZ)+5));
5661 rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
5662 if (rv != CKR_OK) {
5663     .
5664     .
5665 }
5666 rv = C_SignInit(hSession, &signMechanism, hMacKey);
5667 if (rv != CKR_OK) {
5668     .
5669     .
5670 }
5671
5672 ulEncryptedDataLen = sizeof(encryptedData);
5673 rv = C_SignEncryptUpdate(
5674     hSession,
5675     &data[0], BUF_SZ,
5676     encryptedData, &ulEncryptedDataLen);
5677 .
```

```

5678 .
5679 ulEncryptedDataLen = sizeof(encryptedData);
5680 rv = C_SignEncryptUpdate(
5681     hSession,
5682     &data[BUF_SZ], BUF_SZ,
5683     encryptedData, &ulEncryptedDataLen);
5684 .
5685 .
5686
5687 /*
5688  * The last portion of the buffer needs to be handled with
5689  * separate calls to deal with padding issues in ECB mode
5690  */
5691
5692 /* First, complete the signature on the buffer */
5693 rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
5694 .
5695 .
5696 ulMacLen = sizeof(MAC);
5697 rv = C_SignFinal(hSession, MAC, &ulMacLen);
5698 .
5699 .
5700
5701 /* Then pad last part with 3 0x00 bytes, and complete encryption */
5702 for(i=0;i<3;i++)
5703     data[((BUF_SZ*2)+5)+i] = 0x00;
5704
5705 /* Now, get second-to-last piece of ciphertext */
5706 ulEncryptedDataLen = sizeof(encryptedData);
5707 rv = C_EncryptUpdate(
5708     hSession,
5709     &data[BUF_SZ*2], 8,
5710     encryptedData, &ulEncryptedDataLen);
5711 .
5712 .
5713
5714 /* Get last piece of ciphertext (should have length 0, here) */
5715 ulEncryptedDataLen = sizeof(encryptedData);
5716 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
5717 .
5718 .

```


5.17.4 C_DecryptVerifyUpdate

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptVerifyUpdate) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG ulEncryptedPartLen,
    CK_BYTE_PTR pPart,
    CK_ULONG_PTR pulPartLen
);
```

C_DecryptVerifyUpdate continues a multiple-part combined decryption and verification operation, processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

C_DecryptVerifyUpdate uses the convention described in Section 5.2 on producing output. If a **C_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active verification operation.

Decryption and signature operations **MUST** both be active (they **MUST** have been initialized with **C_DecryptInit** and **C_VerifyInit**, respectively). This function may be called any number of times in succession, and may be interspersed with **C_DecryptUpdate** and **C_VerifyUpdate** calls.

Use of **C_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using **C_SignEncryptUpdate**, the "inverse function" of **C_DecryptVerifyUpdate**. This is because when **C_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation and the active encryption operation; however, when **C_DecryptVerifyUpdate** is called, the input passed to the active verifying operation is the *output* of the active decryption operation. This issue comes up only when the mechanism used for decryption performs padding.

In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this ciphertext and verify a signature on the original plaintext thereby obtained.

After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C_DecryptVerifyUpdate**. **C_DecryptVerifyUpdate** returns exactly 16 bytes of plaintext, since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's no more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active decryption and verification operations are linked *only* through the **C_DecryptVerifyUpdate** call, these 2 bytes of plaintext are *not* passed on to the verification mechanism.

A call to **C_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature on *the first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C_VerifyFinal** is called, the last 2 bytes of plaintext get passed into the active verification operation via a **C_VerifyUpdate** call.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active verification operation. *Extreme caution is warranted when using a padded decryption mechanism with C_DecryptVerifyUpdate.*

Return values: `CKR_ARGUMENTS_BAD`, `CKR_BUFFER_TOO_SMALL`, `CKR_CRYPTOKI_NOT_INITIALIZED`, `CKR_DATA_LEN_RANGE`, `CKR_DEVICE_ERROR`, `CKR_DEVICE_MEMORY`, `CKR_DEVICE_REMOVED`, `CKR_ENCRYPTED_DATA_INVALID`, `CKR_ENCRYPTED_DATA_LEN_RANGE`, `CKR_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`.

Example:

```
#define BUF_SZ 512
```

```

5771
5772 CK_SESSION_HANDLE hSession;
5773 CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
5774 CK_BYTE iv[8];
5775 CK_MECHANISM decryptionMechanism = {
5776     CKM_DES_ECB, iv, sizeof(iv)
5777 };
5778 CK_MECHANISM verifyMechanism = {
5779     CKM_DES_MAC, NULL_PTR, 0
5780 };
5781 CK_BYTE encryptedData[(2*BUF_SZ)+8];
5782 CK_BYTE MAC[4];
5783 CK_ULONG ulMacLen;
5784 CK_BYTE data[BUF_SZ];
5785 CK_ULONG ulDataLen, ulLastUpdateSize;
5786 CK_RV rv;
5787
5788 .
5789 .
5790 memset(iv, 0, sizeof(iv));
5791 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
5792 rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
5793 if (rv != CKR_OK) {
5794     .
5795     .
5796 }
5797 rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
5798 if (rv != CKR_OK){
5799     .
5800     .
5801 }
5802
5803 ulDataLen = sizeof(data);
5804 rv = C_DecryptVerifyUpdate(
5805     hSession,
5806     &encryptedData[0], BUF_SZ,
5807     data, &ulDataLen);
5808 .
5809 .
5810 ulDataLen = sizeof(data);
5811 rv = C_DecryptVerifyUpdate(
5812     hSession,
5813     &encryptedData[BUF_SZ], BUF_SZ,

```

```

5814     data, &ulDataLen);
5815     .
5816     .
5817
5818     /*
5819     * The last portion of the buffer needs to be handled with
5820     * separate calls to deal with padding issues in ECB mode
5821     */
5822
5823     /* First, complete the decryption of the buffer */
5824     ulLastUpdateSize = sizeof(data);
5825     rv = C_DecryptUpdate(
5826         hSession,
5827         &encryptedData[BUF_SZ*2], 8,
5828         data, &ulLastUpdateSize);
5829     .
5830     .
5831     /* Get last little piece of plaintext.  Should have length 0 */
5832     ulDataLen = sizeof(data)-ulLastUpdateSize;
5833     rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
5834     if (rv != CKR_OK) {
5835         .
5836         .
5837     }
5838
5839     /* Send last bit of plaintext to verification operation */
5840     rv = C_VerifyUpdate(hSession, data, 5);
5841     if (rv != CKR_OK) {
5842         .
5843         .
5844     }
5845     rv = C_VerifyFinal(hSession, MAC, ulMacLen);
5846     if (rv == CKR_SIGNATURE_INVALID) {
5847         .
5848         .
5849     }

```

5.18 Key management functions

Cryptoki provides the following functions for key management:

5.18.1 C_GenerateKey

```

CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey)(
    CK_SESSION_HANDLE hSession

```

```

5855     CK_MECHANISM_PTR pMechanism,
5856     CK_ATTRIBUTE_PTR pTemplate,
5857     CK_ULONG ulCount,
5858     CK_OBJECT_HANDLE_PTR phKey
5859 );

```

5860 **C_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is
5861 the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template
5862 for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey*
5863 points to the location that receives the handle of the new key or set of domain parameters.

5864 If the generation mechanism is for domain parameter generation, the **CKA_CLASS** attribute will have the
5865 value CKO_DOMAIN_PARAMETERS; otherwise, it will have the value CKO_SECRET_KEY.

5866 Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the
5867 template does not need to supply a key type. If it does supply a key type which is inconsistent with the
5868 generation mechanism, **C_GenerateKey** fails and returns the error code
5869 CKR_TEMPLATE_INCONSISTENT. The CKA_CLASS attribute is treated similarly.

5870 If a call to **C_GenerateKey** cannot support the precise template supplied to it, it will fail and return without
5871 creating an object.

5872 The object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to
5873 CK_TRUE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
5874 assigned (See Section 4.4.1).

5875 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
5876 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
5877 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
5878 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5879 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
5880 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5881 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
5882 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
5883 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
5884 CKR_USER_NOT_LOGGED_IN.

5885 Example:

```

5886 CK_SESSION_HANDLE hSession;
5887 CK_OBJECT_HANDLE hKey;
5888 CK_MECHANISM mechanism = {
5889     CKM_DES_KEY_GEN, NULL_PTR, 0
5890 };
5891 CK_RV rv;
5892
5893 .
5894 .
5895 rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
5896 if (rv == CKR_OK) {
5897     .
5898     .
5899 }

```

5.18.2 C_GenerateKeyPair

```
CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair)(
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_ATTRIBUTE_PTR pPublicKeyTemplate,
    CK_ULONG ulPublicKeyAttributeCount,
    CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
    CK_ULONG ulPrivateKeyAttributeCount,
    CK_OBJECT_HANDLE_PTR phPublicKey,
    CK_OBJECT_HANDLE_PTR phPrivateKey
);
```

C_GenerateKeyPair generates a public/private key pair, creating new key objects. *hSession* is the session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is the number of attributes in the private-key template; *phPublicKey* points to the location that receives the handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new private key.

Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates do not need to supply key types. If one of the templates does supply a key type which is inconsistent with the key generation mechanism, **C_GenerateKeyPair** fails and returns the error code **CKR_TEMPLATE_INCONSISTENT**. The **CKA_CLASS** attribute is treated similarly.

If a call to **C_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return without creating any key objects.

A call to **C_GenerateKeyPair** will never create just one key and return. A call can fail, and create no keys; or it can succeed, and create a matching public/private key pair.

The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL** attributes set to **CK_TRUE**. In addition, the key objects created will both have values for **CKA_UNIQUE_ID** generated and assigned (See Section 4.4.1).

Note carefully the order of the arguments to C_GenerateKeyPair. The last two arguments do not have the same order as they did in the original Cryptoki Version 1.0 document. The order of these two arguments has caused some unfortunate confusion.

Return values: **CKR_ARGUMENTS_BAD**, **CKR_ATTRIBUTE_READ_ONLY**, **CKR_ATTRIBUTE_TYPE_INVALID**, **CKR_ATTRIBUTE_VALUE_INVALID**, **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_CURVE_NOT_SUPPORTED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**, **CKR_DOMAIN_PARAMS_INVALID**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_MECHANISM_INVALID**, **CKR_MECHANISM_PARAM_INVALID**, **CKR_OK**, **CKR_OPERATION_ACTIVE**, **CKR_PIN_EXPIRED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_READ_ONLY**, **CKR_TEMPLATE_INCOMPLETE**, **CKR_TEMPLATE_INCONSISTENT**, **CKR_TOKEN_WRITE_PROTECTED**, **CKR_USER_NOT_LOGGED_IN**.

Example:

```
CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
CK_MECHANISM mechanism = {
    CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
};
CK_ULONG modulusBits = 3072;
CK_BYTE publicExponent[] = { 3 };
```

```

5950 CK_BYTE subject[] = {...};
5951 CK_BYTE id[] = {123};
5952 CK_BBOOL true = CK_TRUE;
5953 CK_ATTRIBUTE publicKeyTemplate[] = {
5954     {CKA_ENCRYPT, &true, sizeof(true)},
5955     {CKA_VERIFY, &true, sizeof(true)},
5956     {CKA_WRAP, &true, sizeof(true)},
5957     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
5958     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof (publicExponent)}
5959 };
5960 CK_ATTRIBUTE privateKeyTemplate[] = {
5961     {CKA_TOKEN, &true, sizeof(true)},
5962     {CKA_PRIVATE, &true, sizeof(true)},
5963     {CKA_SUBJECT, subject, sizeof(subject)},
5964     {CKA_ID, id, sizeof(id)},
5965     {CKA_SENSITIVE, &true, sizeof(true)},
5966     {CKA_DECRYPT, &true, sizeof(true)},
5967     {CKA_SIGN, &true, sizeof(true)},
5968     {CKA_UNWRAP, &true, sizeof(true)}
5969 };
5970 CK_RV rv;
5971
5972 rv = C_GenerateKeyPair(
5973     hSession, &mechanism,
5974     publicKeyTemplate, 5,
5975     privateKeyTemplate, 8,
5976     &hPublicKey, &hPrivateKey);
5977 if (rv == CKR_OK) {
5978     .
5979     .
5980 }

```

5981 5.18.3 C_WrapKey

```

5982 CK_DECLARE_FUNCTION(CK_RV, C_WrapKey) (
5983     CK_SESSION_HANDLE hSession,
5984     CK_MECHANISM_PTR pMechanism,
5985     CK_OBJECT_HANDLE hWrappingKey,
5986     CK_OBJECT_HANDLE hKey,
5987     CK_BYTE_PTR pWrappedKey,
5988     CK_ULONG_PTR pulWrappedKeyLen
5989 );

```

5990 **C_WrapKey** wraps (i.e., encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism*
5991 points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle
5992 of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and
5993 *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

5994 **C_WrapKey** uses the convention described in Section 5.2 on producing output.

5995 The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping,
5996 MUST be CK_TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped MUST also be
5997 CK_TRUE.

5998 If the key to be wrapped cannot be wrapped for some token-specific reason, despite its having its
5999 **CKA_EXTRACTABLE** attribute set to CK_TRUE, then **C_WrapKey** fails with error code
6000 CKR_KEY_NOT_WRAPPABLE. If it cannot be wrapped with the specified wrapping key and mechanism
6001 solely because of its length, then **C_WrapKey** fails with error code CKR_KEY_SIZE_RANGE.

6002 **C_WrapKey** can be used in the following situations:

6003 • To wrap any secret key with a public key that supports encryption and decryption.

6004 • To wrap any secret key with any other secret key. Consideration MUST be given to key size and
6005 mechanism strength or the token may not allow the operation.

6006 • To wrap a private key with any secret key.

6007 Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

6008 To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute
6009 CKA_WRAP_TEMPLATE can be used on the wrapping key to specify an attribute set that will be
6010 compared against the attributes of the key to be wrapped. If all attributes match according to the
6011 C_FindObject rules of attribute matching then the wrap will proceed. The value of this attribute is an
6012 attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If
6013 this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be
6014 checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return
6015 CKR_KEY_HANDLE_INVALID.

6016 Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
6017 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6018 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6019 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
6020 CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
6021 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
6022 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
6023 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
6024 CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE,
6025 CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

6026 Example:

```

6027 CK_SESSION_HANDLE hSession;
6028 CK_OBJECT_HANDLE hWrappingKey, hKey;
6029 CK_MECHANISM mechanism = {
6030     CKM_DES3_ECB, NULL_PTR, 0
6031 };
6032 CK_BYTE wrappedKey[8];
6033 CK_ULONG ulWrappedKeyLen;
6034 CK_RV rv;
6035
6036 .
6037 .
6038 ulWrappedKeyLen = sizeof(wrappedKey);
6039 rv = C_WrapKey(
6040     hSession, &mechanism,
6041     hWrappingKey, hKey,

```

```

6042     wrappedKey, &ulWrappedKeyLen);
6043 if (rv == CKR_OK) {
6044     .
6045     .
6046 }

```

6047 5.18.4 C_UnwrapKey

```

6048 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKey) (
6049     CK_SESSION_HANDLE hSession,
6050     CK_MECHANISM_PTR pMechanism,
6051     CK_OBJECT_HANDLE hUnwrappingKey,
6052     CK_BYTE_PTR pWrappedKey,
6053     CK_ULONG ulWrappedKeyLen,
6054     CK_ATTRIBUTE_PTR pTemplate,
6055     CK_ULONG ulAttributeCount,
6056     CK_OBJECT_HANDLE_PTR phKey
6057 );

```

6058 **C_UnwrapKey** unwraps (i.e. decrypts) a wrapped key, creating a new private key or secret key object.
6059 *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is
6060 the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the
6061 length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the
6062 number of attributes in the template; *phKey* points to the location that receives the handle of the
6063 recovered key.

6064 The **CKA_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports
6065 unwrapping, MUST be CK_TRUE.

6066 The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the
6067 **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by
6068 default set to CK_TRUE.

6069 Some mechanisms may modify, or attempt to modify, the contents of the *pMechanism* structure at the
6070 same time that the key is unwrapped.

6071 If a call to **C_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without
6072 creating any key object.

6073 The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to
6074 CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
6075 assigned (See Section 4.4.1).

6076 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute
6077 **CKA_UNWRAP_TEMPLATE** can be used on the unwrapping key to specify an attribute set that will be
6078 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied
6079 attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute
6080 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
6081 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute
6082 conflict occurs on an attempt to unwrap a key then the function SHALL return
6083 CKR_TEMPLATE_INCONSISTENT.

6084 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
6085 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
6086 CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
6087 CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6088 CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED,
6089 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6090 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
6091 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
6092 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,

6093 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
6094 CKR_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE,
6095 CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN,
6096 CKR_WRAPPED_KEY_INVALID, CKR_WRAPPED_KEY_LEN_RANGE.

6097 Example:

```
6098 CK_SESSION_HANDLE hSession;  
6099 CK_OBJECT_HANDLE hUnwrappingKey, hKey;  
6100 CK_MECHANISM mechanism = {  
6101     CKM_DES3_ECB, NULL_PTR, 0  
6102 };  
6103 CK_BYTE wrappedKey[8] = {...};  
6104 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
6105 CK_KEY_TYPE keyType = CKK_DES;  
6106 CK_BBOOL true = CK_TRUE;  
6107 CK_ATTRIBUTE template[] = {  
6108     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
6109     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
6110     {CKA_ENCRYPT, &true, sizeof(true)},  
6111     {CKA_DECRYPT, &true, sizeof(true)}  
6112 };  
6113 CK_RV rv;  
6114  
6115 .  
6116 .  
6117 rv = C_UnwrapKey(  
6118     hSession, &mechanism, hUnwrappingKey,  
6119     wrappedKey, sizeof(wrappedKey), template, 4, &hKey);  
6120 if (rv == CKR_OK) {  
6121     .  
6122     .  
6123 }
```

6124 5.18.5 C_DeriveKey

```
6125 CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey) (  
6126     CK_SESSION_HANDLE hSession,  
6127     CK_MECHANISM_PTR pMechanism,  
6128     CK_OBJECT_HANDLE hBaseKey,  
6129     CK_ATTRIBUTE_PTR pTemplate,  
6130     CK_ULONG ulAttributeCount,  
6131     CK_OBJECT_HANDLE_PTR phKey  
6132 );
```

6133 **C_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session's
6134 handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the
6135 handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number
6136 of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

6137 The values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and
6138 **CKA_NEVER_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold
6139 for the newly-derived key. See the description of each particular key-derivation mechanism in Section
6140 5.21.2 for any constraints of this type.

6141 If a call to **C_DeriveKey** cannot support the precise template supplied to it, it will fail and return without
6142 creating any key object.

6143 The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to
6144 CK_FALSE. In addition, the object created will have a value for CKA_UNIQUE_ID generated and
6145 assigned (See Section 4.4.1).

6146 To partition the derivation keys so they can only derive a subset of keys the attribute
6147 CKA_DERIVE_TEMPLATE can be used on the derivation keys to specify an attribute set that will be
6148 added to attributes of the key to be derived. If the attributes do not conflict with the user supplied attribute
6149 template, in 'pTemplate', then the derivation will proceed. The value of this attribute is an attribute
6150 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
6151 attribute is not present on the base derivation keys then no additional attributes will be added. If any
6152 attribute conflict occurs on an attempt to derive a key then the function SHALL return
6153 CKR_TEMPLATE_INCONSISTENT.

6154 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
6155 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
6156 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
6157 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
6158 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6159 CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE,
6160 CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
6161 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
6162 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
6163 CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
6164 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

6165 Example:

```
6166 CK_SESSION_HANDLE hSession;  
6167 CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;  
6168 CK_MECHANISM keyPairMechanism = {  
6169     CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0  
6170 };  
6171 CK_BYTE prime[] = {...};  
6172 CK_BYTE base[] = {...};  
6173 CK_BYTE publicKey[128];  
6174 CK_BYTE otherPublicValue[128];  
6175 CK_MECHANISM mechanism = {  
6176     CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)  
6177 };  
6178 CK_ATTRIBUTE template[] = {  
6179     {CKA_VALUE, &publicValue, sizeof(publicValue)}  
6180 };  
6181 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
6182 CK_KEY_TYPE keyType = CKK_DES;  
6183 CK_BBOOL true = CK_TRUE;  
6184 CK_ATTRIBUTE publicKeyTemplate[] = {
```

```

6185     {CKA_PRIME, prime, sizeof(prime)},
6186     {CKA_BASE, base, sizeof(base)}
6187 };
6188 CK_ATTRIBUTE privateKeyTemplate[] = {
6189     {CKA_DERIVE, &true, sizeof(true)}
6190 };
6191 CK_ATTRIBUTE derivedKeyTemplate[] = {
6192     {CKA_CLASS, &keyClass, sizeof(keyClass)},
6193     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6194     {CKA_ENCRYPT, &true, sizeof(true)},
6195     {CKA_DECRYPT, &true, sizeof(true)}
6196 };
6197 CK_RV rv;
6198 .
6199 .
6200 .
6201 rv = C_GenerateKeyPair(
6202     hSession, &keyPairMechanism,
6203     publicKeyTemplate, 2,
6204     privateKeyTemplate, 1,
6205     &hPublicKey, &hPrivateKey);
6206 if (rv == CKR_OK) {
6207     rv = C_GetAttributeValue(hSession, hPublicKey, template, 1);
6208     if (rv == CKR_OK) {
6209         /* Put other guy's public value in otherPublicValue */
6210         .
6211         .
6212         rv = C_DeriveKey(
6213             hSession, &mechanism,
6214             hPrivateKey, derivedKeyTemplate, 4, &hKey);
6215         if (rv == CKR_OK) {
6216             .
6217             .
6218         }
6219     }
6220 }

```

5.19 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

5.19.1 C_SeedRandom

```

CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom) (
    CK_SESSION_HANDLE hSession,

```

```

6226     CK_BYTE_PTR pSeed,
6227     CK_ULONG ulSeedLen
6228 );

```

6229 **C_SeedRandom** mixes additional seed material into the token's random number generator. *hSession* is
6230 the session's handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed
6231 material.

6232 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6233 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6234 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6235 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6236 CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED,
6237 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6238 Example: see **C_GenerateRandom**.

6239 5.19.2 C_GenerateRandom

```

6240 CK_DECLARE_FUNCTION(CK_RV, C_GenerateRandom) (
6241     CK_SESSION_HANDLE hSession,
6242     CK_BYTE_PTR pRandomData,
6243     CK_ULONG ulRandomLen
6244 );

```

6245 **C_GenerateRandom** generates random or pseudo-random data. *hSession* is the session's handle;
6246 *pRandomData* points to the location that receives the random data; and *ulRandomLen* is the length in
6247 bytes of the random or pseudo-random data to be generated.

6248 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6249 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6250 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6251 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_RANDOM_NO_RNG,
6252 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6253 Example:

```

6254 CK_SESSION_HANDLE hSession;
6255 CK_BYTE seed[] = {...};
6256 CK_BYTE randomData[] = {...};
6257 CK_RV rv;
6258
6259 .
6260 .
6261 rv = C_SeedRandom(hSession, seed, sizeof(seed));
6262 if (rv != CKR_OK) {
6263     .
6264     .
6265 }
6266 rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));
6267 if (rv == CKR_OK) {
6268     .
6269     .
6270 }

```

5.20 Parallel function management functions

Cryptoki provides the following functions for managing parallel execution of cryptographic functions. These functions exist only for backwards compatibility.

5.20.1 C_GetFunctionStatus

```
CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus)(  
    CK_SESSION_HANDLE hSession  
);
```

In previous versions of Cryptoki, **C_GetFunctionStatus** obtained the status of a function running in parallel with an application. Now, however, **C_GetFunctionStatus** is a legacy function which should simply return the value **CKR_FUNCTION_NOT_PARALLEL**.

Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_FUNCTION_FAILED**, **CKR_FUNCTION_NOT_PARALLEL**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_CLOSED**.

5.20.2 C_CancelFunction

```
CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction)(  
    CK_SESSION_HANDLE hSession  
);
```

In previous versions of Cryptoki, **C_CancelFunction** cancelled a function running in parallel with an application. Now, however, **C_CancelFunction** is a legacy function which should simply return the value **CKR_FUNCTION_NOT_PARALLEL**.

Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_FUNCTION_FAILED**, **CKR_FUNCTION_NOT_PARALLEL**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_SESSION_HANDLE_INVALID**, **CKR_SESSION_CLOSED**.

5.21 Callback functions

Cryptoki sessions can use function pointers of type **CK_NOTIFY** to notify the application of certain events.

5.21.1 Surrender callbacks

Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions; decryption functions; message digesting functions; signing and MACing functions; functions for verifying signatures and MACs; dual-purpose cryptographic functions; key management functions; random number generation functions) executing in Cryptoki sessions can periodically surrender control to the application who called them if the session they are executing in had a notification callback function associated with it when it was opened. They do this by calling the session's callback with the arguments (hSession, CKN_SURRENDER, pApplication), where hSession is the session's handle and pApplication was supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the value **CKR_OK** (to indicate that Cryptoki should continue executing the function) or the value **CKR_CANCEL** (to indicate that Cryptoki should abort execution of the function). Of course, before returning one of these values, the callback function can perform some computation, if desired.

A typical use of a surrender callback might be to give an application user feedback during a lengthy key pair generation operation. Each time the application receives a callback, it could display an additional "." to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the key pair generation operation (probably by returning the value **CKR_CANCEL**) if the user hit <ESCAPE>.

A Cryptoki library is *not required* to make any surrender callbacks.

6314 **5.21.2 Vendor-defined callbacks**

6315 Library vendors can also define additional types of callbacks. Because of this extension capability,
6316 application-supplied notification callback routines should examine each callback they receive, and if they
6317 are unfamiliar with the type of that callback, they should immediately give control back to the library by
6318 returning with the value CKR_OK.

6 Mechanisms

6.1 RSA

Table 32, 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_SHA224_RSA_PKCS		✓					
CKM_SHA256_RSA_PKCS		✓					
CKM_SHA384_RSA_PKCS		✓					
CKM_SHA512_RSA_PKCS		✓					
CKM_SHA1_RSA_PKCS_PSS		✓					
CKM_SHA224_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		✓					

6.1.1 Definitions

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

Mechanisms:

CKM_RSA_PKCS_KEY_PAIR_GEN

CKM_RSA_PKCS

CKM_RSA_9796

6329 CKM_RSA_X_509
 6330 CKM_MD2_RSA_PKCS
 6331 CKM_MD5_RSA_PKCS
 6332 CKM_SHA1_RSA_PKCS
 6333 CKM_SHA224_RSA_PKCS
 6334 CKM_SHA256_RSA_PKCS
 6335 CKM_SHA384_RSA_PKCS
 6336 CKM_SHA512_RSA_PKCS
 6337 CKM_RIPEMD128_RSA_PKCS
 6338 CKM_RIPEMD160_RSA_PKCS
 6339 CKM_RSA_PKCS_OAEP
 6340 CKM_RSA_X9_31_KEY_PAIR_GEN
 6341 CKM_RSA_X9_31
 6342 CKM_SHA1_RSA_X9_31
 6343 CKM_RSA_PKCS_PSS
 6344 CKM_SHA1_RSA_PKCS_PSS
 6345 CKM_SHA224_RSA_PKCS_PSS
 6346 CKM_SHA256_RSA_PKCS_PSS
 6347 CKM_SHA512_RSA_PKCS_PSS
 6348 CKM_SHA384_RSA_PKCS_PSS
 6349 CKM_RSA_PKCS_TPM_1_1
 6350 CKM_RSA_PKCS_OAEP_TPM_1_1
 6351 CKM_RSA_AES_KEY_WRAP
 6352 CKM_SHA3_224_RSA_PKCS
 6353 CKM_SHA3_256_RSA_PKCS
 6354 CKM_SHA3_384_RSA_PKCS
 6355 CKM_SHA3_512_RSA_PKCS
 6356 CKM_SHA3_224_RSA_PKCS_PSS
 6357 CKM_SHA3_256_RSA_PKCS_PSS
 6358 CKM_SHA3_384_RSA_PKCS_PSS
 6359 CKM_SHA3_512_RSA_PKCS_PSS
 6360

6361 6.1.2 RSA public key objects

6362 RSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_RSA**) hold RSA public keys.
 6363 The following table defines the RSA public key object attributes, in addition to the common attributes
 6364 defined for this object class:

6365 *Table 33, 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

6366 Refer to Table 11 for footnotes

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

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_RSA;
CK_UTF8CHAR label[] = "An RSA public key object";
CK_BYTE modulus[] = {...};
CK_BYTE exponent[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_WRAP, &true, sizeof(true)},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_MODULUS, modulus, sizeof(modulus)},
    {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
};
```

6.1.3 RSA private key objects

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

Table 34, RSA Private Key Object Attributes

Attribute	Data type	Meaning
CKA_MODULUS ^{1,4,6}	Big integer	Modulus n
CKA_PUBLIC_EXPONENT ^{1,4,6}	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 $q^{-1} \bmod p$

Refer to Table 11 for footnotes

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

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

Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an RSA private key, it stores whichever of the fields in Table 34 it keeps track of. Later, if an application asks for the values of the key's various attributes, Cryptoki supplies values only for attributes whose values it can obtain (i.e., if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA private keys which are not actually stored on the token. *E.g.*, if a particular token stores

6405 values only for the **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then
6406 Cryptoki is certainly *able* to report values for all the attributes above (since they can all be computed
6407 efficiently from these three values). However, a Cryptoki implementation may or may not actually do this
6408 extra computation. The only attributes from Table 34 for which a Cryptoki implementation is *required* to
6409 be able to return values are **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT** and
6410 **CKA_PRIVATE_EXPONENT**. A token SHOULD also be able to return **CKA_PUBLIC_KEY_INFO** for an
6411 RSA private key.

6412 If an RSA private key object is created on a token, and more attributes from Table 34 are supplied to the
6413 object creation call than are supported by the token, the extra attributes are likely to be thrown away. If
6414 an attempt is made to create an RSA private key object on a token with insufficient attributes for that
6415 particular token, then the object creation call fails and returns CKR_TEMPLATE_INCOMPLETE.

6416 Note that when generating an RSA private key, there is no **CKA_MODULUS_BITS** attribute specified.
6417 This is because RSA private keys are only generated as part of an RSA key *pair*, and the
6418 **CKA_MODULUS_BITS** attribute for the pair is specified in the template for the RSA public key.

6419 The following is a sample template for creating an RSA private key object:

```
6420     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
6421     CK_KEY_TYPE keyType = CKK_RSA;
6422     CK_UTF8CHAR label[] = "An RSA private key object";
6423     CK_BYTE subject[] = {...};
6424     CK_BYTE id[] = {123};
6425     CK_BYTE modulus[] = {...};
6426     CK_BYTE publicExponent[] = {...};
6427     CK_BYTE privateExponent[] = {...};
6428     CK_BYTE prime1[] = {...};
6429     CK_BYTE prime2[] = {...};
6430     CK_BYTE exponent1[] = {...};
6431     CK_BYTE exponent2[] = {...};
6432     CK_BYTE coefficient[] = {...};
6433     CK_BBOOL true = CK_TRUE;
6434     CK_ATTRIBUTE template[] = {
6435         {CKA_CLASS, &class, sizeof(class)},
6436         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6437         {CKA_TOKEN, &true, sizeof(true)},
6438         {CKA_LABEL, label, sizeof(label)-1},
6439         {CKA_SUBJECT, subject, sizeof(subject)},
6440         {CKA_ID, id, sizeof(id)},
6441         {CKA_SENSITIVE, &true, sizeof(true)},
6442         {CKA_DECRYPT, &true, sizeof(true)},
6443         {CKA_SIGN, &true, sizeof(true)},
6444         {CKA_MODULUS, modulus, sizeof(modulus)},
6445         {CKA_PUBLIC_EXPONENT, publicExponent,
6446             sizeof(publicExponent)},
6447         {CKA_PRIVATE_EXPONENT, privateExponent,
6448             sizeof(privateExponent)},
6449         {CKA_PRIME_1, prime1, sizeof(prime1)},
6450         {CKA_PRIME_2, prime2, sizeof(prime2)},
6451         {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
6452         {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
6453         {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
```

6454 } ;

6455 6.1.4 PKCS #1 RSA key pair generation

6456 The PKCS #1 RSA key pair generation mechanism, denoted **CKM_RSA_PKCS_KEY_PAIR_GEN**, is a
6457 key pair generation mechanism based on the RSA public-key cryptosystem, as defined in PKCS #1.

6458 It does not have a parameter.

6459 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
6460 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
6461 template for the public key. The **CKA_PUBLIC_EXPONENT** may be omitted in which case the
6462 mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537).
6463 Specific implementations may use a random value or an alternative default if 0x10001 cannot be used by
6464 the token.

6465 Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error
6466 if this attribute is omitted from the template. Experience has shown that many implementations of 2.11
6467 and prior did allow the **CKA_PUBLIC_EXPONENT** attribute to be omitted from the template, and
6468 behaved as described above. The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**,
6469 **CKA_MODULUS**, and **CKA_PUBLIC_EXPONENT** attributes to the new public key.

6470 **CKA_PUBLIC_EXPONENT** will be copied from the template if supplied.

6471 **CKR_TEMPLATE_INCONSISTENT** shall be returned if the implementation cannot use the supplied
6472 exponent value. It contributes the **CKA_CLASS** and **CKA_KEY_TYPE** attributes to the new private key; it
6473 may also contribute some of the following attributes to the new private key: **CKA_MODULUS**,
6474 **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**,
6475 **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**. Other attributes supported by the
6476 RSA public and private key types (specifically, the flags indicating which functions the keys support) may
6477 also be specified in the templates for the keys, or else are assigned default initial values.

6478 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6479 specify the supported range of RSA modulus sizes, in bits.

6480 6.1.5 X9.31 RSA key pair generation

6481 The X9.31 RSA key pair generation mechanism, denoted **CKM_RSA_X9_31_KEY_PAIR_GEN**, is a key
6482 pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

6483 It does not have a parameter.

6484 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
6485 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
6486 template for the public key.

6487 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**, and
6488 **CKA_PUBLIC_EXPONENT** attributes to the new public key. It contributes the **CKA_CLASS** and
6489 **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes
6490 to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**,
6491 **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**.
6492 Other attributes supported by the RSA public and private key types (specifically, the flags indicating which
6493 functions the keys support) may also be specified in the templates for the keys, or else are assigned
6494 default initial values. Unlike the **CKM_RSA_PKCS_KEY_PAIR_GEN** mechanism, this mechanism is
6495 guaranteed to generate *p* and *q* values, **CKA_PRIME_1** and **CKA_PRIME_2** respectively, that meet the
6496 strong primes requirement of X9.31.

6497 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6498 specify the supported range of RSA modulus sizes, in bits.

6499 6.1.6 PKCS #1 v1.5 RSA

6500 The PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS**, is a multi-purpose mechanism based
6501 on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5. It supports

single-part encryption and decryption; single-part signatures and verification with and without message recovery; key wrapping; and key unwrapping. This mechanism corresponds only to the part of PKCS #1 v1.5 that involves RSA; it does not compute a message digest or a DigestInfo encoding as specified for the md2withRSAEncryption and md5withRSAEncryption algorithms in PKCS #1 v1.5 .

This mechanism does not have a parameter.

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

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

Table 35, 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

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 RSA modulus sizes, in bits.

6.1.7 PKCS #1 RSA OAEP mechanism parameters

◆ CK_RSA_PKCS_MGF_TYPE; CK_RSA_PKCS_MGF_TYPE_PTR

CK_RSA_PKCS_MGF_TYPE is used to indicate the Mask Generation Function (MGF) applied to a message block when formatting a message block for the PKCS #1 OAEP encryption scheme or the PKCS #1 PSS signature scheme. It is defined as follows:

```
typedef CK_ULONG CK_RSA_PKCS_MGF_TYPE;
```

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

Table 36, 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

6531 **CK_RSA_PKCS_MGF_TYPE_PTR** is a pointer to a **CK_RSA_PKCS_MGF_TYPE**.

6532 ♦ **CK_RSA_PKCS_OAEP_SOURCE_TYPE;**
6533 **CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR**

6534 **CK_RSA_PKCS_OAEP_SOURCE_TYPE** is used to indicate the source of the encoding parameter
6535 when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

6536 `typedef CK_ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;`

6537

6538 The following encoding parameter sources are defined in PKCS #1. The following table lists the defined
6539 sources along with the corresponding data type for the *pSourceData* field in the
6540 **CK_RSA_PKCS_OAEP_PARAMS** structure defined below.

6541 *Table 37, 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.

6542 **CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR** is a pointer to a
6543 **CK_RSA_PKCS_OAEP_SOURCE_TYPE**.

6544 ♦ **CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR**

6545 **CK_RSA_PKCS_OAEP_PARAMS** is a structure that provides the parameters to the
6546 **CKM_RSA_PKCS_OAEP** mechanism. The structure is defined as follows:

```
6547     typedef struct CK_RSA_PKCS_OAEP_PARAMS {
6548         CK_MECHANISM_TYPE          hashAlg;
6549         CK_RSA_PKCS_MGF_TYPE       mgf;
6550         CK_RSA_PKCS_OAEP_SOURCE_TYPE source;
6551         CK_VOID_PTR                pSourceData;
6552         CK_ULONG                   ulSourceDataLen;
6553     } CK_RSA_PKCS_OAEP_PARAMS;
```

6554

6555 The fields of the structure have the following meanings:

6556 `hashAlg` mechanism ID of the message digest algorithm used to calculate
6557 the digest of the encoding parameter
6558 `mgf` mask generation function to use on the encoded block

6559 source source of the encoding parameter
6560 pSourceData data used as the input for the encoding parameter source
6561 ulSourceDataLen length of the encoding parameter source input
6562 **CK_RSA_PKCS_OAEP_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_OAEP_PARAMS**.
6563

6564 **6.1.8 PKCS #1 RSA OAEP**

6565 The PKCS #1 RSA OAEP mechanism, denoted **CKM_RSA_PKCS_OAEP**, is a multi-purpose
6566 mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1.
6567 It supports single-part encryption and decryption; key wrapping; and key unwrapping.

6568 It has a parameter, a **CK_RSA_PKCS_OAEP_PARAMS** structure.

6569 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
6570 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
6571 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
6572 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
6573 key, except the key length; the application must convey these separately. In particular, the mechanism
6574 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
6575 to the recovered key during unwrapping; other attributes must be specified in the template.

6576 Constraints on key types and the length of the data are summarized in the following table. For encryption
6577 and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the
6578 length in bytes of the RSA modulus, and *hLen* is the output length of the message digest algorithm
6579 specified by the *hashAlg* field of the **CK_RSA_PKCS_OAEP_PARAMS** structure.

6580 *Table 38, PKCS #1 RSA OAEP: Key And Data Length*

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

6581 ¹ Single-part operations only.

6582 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6583 specify the supported range of RSA modulus sizes, in bits.

6584 **6.1.9 PKCS #1 RSA PSS mechanism parameters**

6585 **◆ CK_RSA_PKCS_PSS_PARAMS; CK_RSA_PKCS_PSS_PARAMS_PTR**

6586 **CK_RSA_PKCS_PSS_PARAMS** is a structure that provides the parameters to the
6587 **CKM_RSA_PKCS_PSS** mechanism. The structure is defined as follows:

```
6588     typedef struct CK_RSA_PKCS_PSS_PARAMS {  
6589         CK_MECHANISM_TYPE    hashAlg;  
6590         CK_RSA_PKCS_MGF_TYPE mgf;  
6591         CK_ULONG              sLen;  
6592     } CK_RSA_PKCS_PSS_PARAMS;
```

6593
6594 The fields of the structure have the following meanings:

6595 hashAlg hash algorithm used in the PSS encoding; if the signature
6596 mechanism does not include message hashing, then this value must
6597 be the mechanism used by the application to generate the message
6598 hash; if the signature mechanism includes hashing, then this value
6599 must match the hash algorithm indicated by the signature
6600 mechanism

6601 mgf mask generation function to use on the encoded block

6602 sLen length, in bytes, of the salt value used in the PSS encoding; typical
6603 values are the length of the message hash and zero

6604 **CK_RSA_PKCS_PSS_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_PSS_PARAMS**.

6605 6.1.10 PKCS #1 RSA PSS

6606 The PKCS #1 RSA PSS mechanism, denoted **CKM_RSA_PKCS_PSS**, is a mechanism based on the
6607 RSA public-key cryptosystem and the PSS block format defined in PKCS #1. It supports single-part
6608 signature generation and verification without message recovery. This mechanism corresponds only to the
6609 part of PKCS #1 that involves block formatting and RSA, given a hash value; it does not compute a hash
6610 value on the message to be signed.

6611 It has a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must be less than or
6612 equal to $k^*-2\cdot hLen$ and *hLen* is the length of the input to the C_Sign or C_Verify function. k^* is the length
6613 in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple
6614 of 8, in which case k^* is one less than the length in bytes of the RSA modulus.

6615 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*
6616 is the length in bytes of the RSA.

6617 Table 39, 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</i> , <i>k</i>	N/A

6618 1 Single-part operations only.

6619 2 Data length, signature length.

6620 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
6621 specify the supported range of RSA modulus sizes, in bits.

6622 6.1.11 ISO/IEC 9796 RSA

6623 The ISO/IEC 9796 RSA mechanism, denoted **CKM_RSA_9796**, is a mechanism for single-part
6624 signatures and verification with and without message recovery based on the RSA public-key
6625 cryptosystem and the block formats defined in ISO/IEC 9796 and its annex A.

6626 This mechanism processes only byte strings, whereas ISO/IEC 9796 operates on bit strings. Accordingly,
6627 the following transformations are performed:

- 6628 • Data is converted between byte and bit string formats by interpreting the most-significant bit of the
6629 leading byte of the byte string as the leftmost bit of the bit string, and the least-significant bit of the
6630 trailing byte of the byte string as the rightmost bit of the bit string (this assumes the length in bits of
6631 the data is a multiple of 8).
- 6632 • A signature is converted from a bit string to a byte string by padding the bit string on the left with 0 to
6633 7 zero bits so that the resulting length in bits is a multiple of 8, and converting the resulting bit string
6634 as above; it is converted from a byte string to a bit string by converting the byte string as above, and
6635 removing bits from the left so that the resulting length in bits is the same as that of the RSA modulus.

6636 This mechanism does not have a parameter.

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

Table 40, 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	$\leq \lfloor k/2 \rfloor$

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 RSA modulus sizes, in bits.

6.1.12 X.509 (raw) RSA

The X.509 (raw) RSA mechanism, denoted **CKM_RSA_X_509**, is a multi-purpose mechanism based on the RSA public-key cryptosystem. It supports single-part encryption and decryption; single-part signatures and verification with and without message recovery; key wrapping; and key unwrapping. All these operations are based on so-called “raw” RSA, as assumed in X.509.

“Raw” RSA as defined here encrypts a byte string by converting it to an integer, most-significant byte first, applying “raw” RSA exponentiation, and converting the result to a byte string, most-significant byte first. The input string, considered as an integer, must be less than the modulus; the output string is also less than the modulus.

This mechanism does not have a parameter.

This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped; similarly for unwrapping. The mechanism does not wrap the key type, key length, or any other information about the key; the application must convey these separately, and supply them when unwrapping the key.

Unfortunately, X.509 does not specify how to perform padding for RSA encryption. For this mechanism, padding should be performed by prepending plaintext data with 0-valued bytes. In effect, to encrypt the 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 be less than the RSA modulus. The k -byte ciphertext (k is the length in bytes of the RSA modulus) is produced by raising P to the RSA public exponent modulo the RSA modulus. Decryption of a k -byte ciphertext C is accomplished by raising C to the RSA private exponent modulo the RSA modulus, and returning the resulting value as a sequence of exactly k bytes. If the resulting plaintext is to be used to produce an unwrapped key, then however many bytes are specified in the template for the length of the key are taken *from the end* of this sequence of bytes.

Technically, the above procedures may differ very slightly from certain details of what is specified in X.509.

Executing cryptographic operations using this mechanism can result in the error returns **CKR_DATA_INVALID** (if plaintext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus) and **CKR_ENCRYPTED_DATA_INVALID** (if ciphertext is supplied which has the same length as the RSA modulus and is numerically at least as large as the modulus).

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

Table 41, 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)

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 RSA modulus sizes, in bits.

This mechanism is intended for compatibility with applications that do not follow the PKCS #1 or ISO/IEC 9796 block formats.

6.1.13 ANSI X9.31 RSA

The ANSI X9.31 RSA mechanism, denoted **CKM_RSA_X9_31**, is a mechanism for single-part signatures and verification without message recovery based on the RSA public-key cryptosystem and the block formats defined in ANSI X9.31.

This mechanism applies the header and padding fields of the hash encapsulation. The trailer field must be applied by the application.

This mechanism processes only byte strings, whereas ANSI X9.31 operates on bit strings. Accordingly, 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.

This mechanism does not have a parameter.

Constraints on key types and the length of input and output data 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.

Table 42, 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

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 RSA modulus sizes, in bits.

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

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

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

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

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

None of these mechanisms has 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 the PKCS #1 v1.5 RSA signature with MD2 and PKCS #1 v1.5 RSA signature with MD5 mechanisms, k must be at least 27; for the PKCS #1 v1.5 RSA signature with SHA-1 mechanism, k must be at least 31, and so on for other underlying hash functions, where the minimum is always 11 bytes more than the length of the hash value.

Table 43, 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	k	block type 01
C_Verify	RSA public key	any, k^2	N/A	block type 01

² Data length, signature length.

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

6.1.15 PKCS #1 v1.5 RSA signature with SHA-224

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

6.1.16 PKCS #1 RSA PSS signature with SHA-224

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

6.1.17 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 $k^* - 2 \cdot hLen$ 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 44, 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.

6.1.18 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.

6.1.19 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_PKCS_PSS**, **CKM_SHA3_256_RSA_PKCS_PSS**, **CKM_SHA3_384_RSA_PKCS_PSS**, and **CKM_SHA3_512_RSA_PKCS_PSS** respectively, performs similarly as the other **CKM_SHAX_RSA_PKCS_PSS** mechanisms but uses the corresponding SHA-3 hash functions.

6.1.20 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.

Table 45, 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	<i>k</i>
C_Verify	RSA public key	any, k^2	N/A

² Data length, signature length.

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

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

The TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS_TPM_1_1**, is a multi-use mechanism based on the RSA public-key cryptosystem and the block formats initially defined in PKCS #1 v1.5, with additional formatting rules defined in TCGA TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.

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

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

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

Table 46, 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	$\leq k-11-5$	<i>k</i>
C_Decrypt ¹	RSA private key	<i>k</i>	$\leq k-11-5$
C_WrapKey	RSA public key	$\leq k-11-5$	<i>k</i>
C_UnwrapKey	RSA private key	<i>k</i>	$\leq k-11-5$

¹ Single-part operations only.

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

6.1.22 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP

The TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP mechanism, denoted **CKM_RSA_PKCS_OAEP_TPM_1_1**, is a multi-purpose mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in PKCS #1, with additional formatting defined in TCGA TPM Specification Version 1.1b. Additional formatting rules remained the same in TCG TPM Specification 1.2. The mechanism supports single-part encryption and decryption; key wrapping; and key unwrapping.

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

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

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

Table 47, 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	$\leq k-2-40-5$

¹ Single-part operations only.

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

6.1.23 RSA AES KEY WRAP

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

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

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

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

For wrapping, the mechanism -

- Generates a temporary random AES key of *ulAESKeyBits* length. This key is not accessible to the user - no handle is returned.
- Wraps the AES key with the wrapping RSA key using **CKM_RSA_PKCS_OAEP** with parameters of *OAEPParams*.
- Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- Zeroizes the temporary AES key
- Concatenates two wrapped keys and outputs the concatenated blob. The first is the wrapped AES key, and the second is the wrapped target key.

The private target key will be encoded as defined in section 6.7.

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

For unwrapping, the mechanism -

- Splits the input into two parts. The first is the wrapped AES key, and the second is the wrapped target key. The length of the first part is equal to the length of the unwrapping RSA key.
- Un-wraps the temporary AES key from the first part with the private RSA key using **CKM_RSA_PKCS_OAEP** with parameters of *OAEPParams*.
- Un-wraps the target key from the second part with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- Zeroizes the temporary AES key.
- Returns the handle to the newly unwrapped target key.

Table 48, 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						✓	
¹ SR = SignRecover, VR = VerifyRecover							

6.1.24 RSA AES KEY WRAP mechanism parameters

◆ **CK_RSA_AES_KEY_WRAP_PARAMS; CK_RSA_AES_KEY_WRAP_PARAMS_PTR**

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

```
typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
    CK_ULONG                ulAESKeyBits;
    CK_RSA_PKCS_OAEP_PARAMS_PTR pOAEPParams;
} CK_RSA_AES_KEY_WRAP_PARAMS;
```

The fields of the structure have the following meanings:

ulAESKeyBits	length of the temporary AES key in bits. Can be only 128, 192 or 256.
pOAEPParams	pointer to the parameters of the temporary AES key wrapping. See also the description of PKCS #1 RSA OAEP mechanism parameters.

CK_RSA_AES_KEY_WRAP_PARAMS_PTR is a pointer to a **CK_RSA_AES_KEY_WRAP_PARAMS**.

6.1.25 FIPS 186-4

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

6.2 DSA

Table 49, 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_PROBABILISTIC_PARAMETER_GEN					✓		
CKM_DSA_SHAWA_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		✓					

6.2.1 Definitions

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

Mechanisms:

- CKM_DSA_KEY_PAIR_GEN
- CKM_DSA
- 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
- CKM_DSA_PARAMETER_GEN
- CKM_DSA_PROBABILISTIC_PARAMETER_GEN
- CKM_DSA_SHAWA_TAYLOR_PARAMETER_GEN
- CKM_DSA_FIPS_G_GEN

◆ CK_DSA_PARAMETER_GEN_PARAM

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

CK_DSA_PARAMETER_GEN_PARAM_PTR is a pointer to a CK_DSA_PARAMETER_GEN_PARAM.

```
typedef struct CK_DSA_PARAMETER_GEN_PARAM {
    CK_MECHANISM_TYPE    hash;
    CK_BYTE_PTR          pSeed;
    CK_ULONG              ulSeedLen;
    CK_ULONG              ulIndex;
} CK_DSA_PARAMETER_GEN_PARAM;
```

The fields of the structure have the following meanings:

hash	Mechanism value for the base hash used in PQG generation, Valid values are CKM_SHA_1, CKM_SHA224, CKM_SHA256, CKM_SHA384, CKM_SHA512.
pSeed	Seed value used to generate PQ and G. This value is returned by CKM_DSA_PROBABILISTIC_PARAMETER_GEN, CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN, and passed into CKM_DSA_FIPS_G_GEN.
ulSeedLen	Length of seed value.
ulIndex	Index value for generating G. Input for CKM_DSA_FIPS_G_GEN. Ignored by CKM_DSA_PROBABILISTIC_PARAMETER_GEN and CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN.

6.2.2 DSA public key objects

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

Table 50, 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

Refer to Table 11 for footnotes

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

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DSA;
CK_UTF8CHAR label[] = "A DSA public key object";
CK_BYTE prime[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE base[] = {...};
```



```

6962     CK_BYTE value[] = {...};
6963     CK_BBOOL true = CK_TRUE;
6964     CK_ATTRIBUTE template[] = {
6965         {CKA_CLASS, &class, sizeof(class)},
6966         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
6967         {CKA_TOKEN, &true, sizeof(true)},
6968         {CKA_LABEL, label, sizeof(label)-1},
6969         {CKA_PRIME, prime, sizeof(prime)},
6970         {CKA_SUBPRIME, subprime, sizeof(subprime)},
6971         {CKA_BASE, base, sizeof(base)},
6972         {CKA_VALUE, value, sizeof(value)}
6973     };
6974

```

6.2.3 DSA Key Restrictions

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

- Prime: 1024 bits, Subprime: 160
- Prime: 2048 bits, Subprime: 224
- Prime: 2048 bits, Subprime: 256
- Prime: 3072 bits, Subprime: 256

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

6.2.4 DSA private key objects

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

Table 51, 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

Refer to Table 11 for footnotes

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

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

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

```

6996     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
6997     CK_KEY_TYPE keyType = CKK_DSA;
6998     CK_UTF8CHAR label[] = "A DSA private key object";
6999     CK_BYTE subject[] = {...};
7000     CK_BYTE id[] = {123};

```

```
7001 CK_BYTE prime[] = {...};
7002 CK_BYTE subprime[] = {...};
7003 CK_BYTE base[] = {...};
7004 CK_BYTE value[] = {...};
7005 CK_BBOOL true = CK_TRUE;
7006 CK_ATTRIBUTE template[] = {
7007     {CKA_CLASS, &class, sizeof(class)},
7008     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7009     {CKA_TOKEN, &true, sizeof(true)},
7010     {CKA_LABEL, label, sizeof(label)-1},
7011     {CKA_SUBJECT, subject, sizeof(subject)},
7012     {CKA_ID, id, sizeof(id)},
7013     {CKA_SENSITIVE, &true, sizeof(true)},
7014     {CKA_SIGN, &true, sizeof(true)},
7015     {CKA_PRIME, prime, sizeof(prime)},
7016     {CKA_SUBPRIME, subprime, sizeof(subprime)},
7017     {CKA_BASE, base, sizeof(base)},
7018     {CKA_VALUE, value, sizeof(value)}
7019 };
```

7020 **6.2.5 DSA domain parameter objects**

7021 DSA domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_DSA**) hold
7022 DSA domain parameters. The following table defines the DSA domain parameter object attributes, in
7023 addition to the common attributes defined for this object class:

7024 *Table 52, DSA Domain Parameter Object Attributes*

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

7025 Refer to Table 11 for footnotes

7026 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
7027 parameters”. See FIPS PUB 186-4 for more information on DSA domain parameters.

7028 To ensure backwards compatibility, if **CKA_SUBPRIME_BITS** is not specified for a call to
7029 **C_GenerateKey**, it takes on a default based on the value of **CKA_PRIME_BITS** as follows:

- 7030
- If **CKA_PRIME_BITS** is less than or equal to 1024 then **CKA_SUBPRIME_BITS** shall be 160 bits
 - If **CKA_PRIME_BITS** equals 2048 then **CKA_SUBPRIME_BITS** shall be 224 bits
 - If **CKA_PRIME_BITS** equals 3072 then **CKA_SUBPRIME_BITS** shall be 256 bits
- 7031
- 7032
- 7033

7034 The following is a sample template for creating a DSA domain parameter object:

```
7035 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
7036 CK_KEY_TYPE keyType = CKK_DSA;
7037 CK_UTF8CHAR label[] = "A DSA domain parameter object";
7038 CK_BYTE prime[] = {...};
7039 CK_BYTE subprime[] = {...};
7040 CK_BYTE base[] = {...};
```

```

7041     CK_BBOOL true = CK_TRUE;
7042     CK_ATTRIBUTE template[] = {
7043         {CKA_CLASS, &class, sizeof(class)},
7044         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7045         {CKA_TOKEN, &true, sizeof(true)},
7046         {CKA_LABEL, label, sizeof(label)-1},
7047         {CKA_PRIME, prime, sizeof(prime)},
7048         {CKA_SUBPRIME, subprime, sizeof(subprime)},
7049         {CKA_BASE, base, sizeof(base)},
7050     };

```

7051 6.2.6 DSA key pair generation

7052 The DSA key pair generation mechanism, denoted **CKM_DSA_KEY_PAIR_GEN**, is a key pair generation
7053 mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

7054 This mechanism does not have a parameter.

7055 The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as
7056 specified in the **CKA_PRIME**, **CKA_SUBPRIME**, and **CKA_BASE** attributes of the template for the public
7057 key.

7058 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
7059 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**, **CKA_BASE**, and
7060 **CKA_VALUE** attributes to the new private key. Other attributes supported by the DSA public and private
7061 key types (specifically, the flags indicating which functions the keys support) may also be specified in the
7062 templates for the keys, or else are assigned default initial values.

7063 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7064 specify the supported range of DSA prime sizes, in bits.

7065 6.2.7 DSA domain parameter generation

7066 The DSA domain parameter generation mechanism, denoted **CKM_DSA_PARAMETER_GEN**, is a
7067 domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB
7068 186-2.

7069 This mechanism does not have a parameter.

7070 The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in
7071 the **CKA_PRIME_BITS** attribute of the template.

7072 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
7073 **CKA_BASE** and **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the DSA
7074 domain parameter types may also be specified in the template, or else are assigned default initial values.

7075 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7076 specify the supported range of DSA prime sizes, in bits.

7077 6.2.8 DSA probabilistic domain parameter generation

7078 The DSA probabilistic domain parameter generation mechanism, denoted
7079 **CKM_DSA_PROBABILISTIC_PARAMETER_GEN**, is a domain parameter generation mechanism based
7080 on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.1 Generation and
7081 Validation of Probable Primes..

7082 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
7083 returns the seed (*pSeed*) and the length (*ulSeedLen*).

7084 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
7085 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
7086 specified in the **CKA_SUBPRIME_BITS** attribute of the template.

7087 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
7088 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
7089 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
7090 template, or else are assigned default initial values.
7091 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7092 specify the supported range of DSA prime sizes, in bits.

7093 6.2.9 DSA Shawe-Taylor domain parameter generation

7094 The DSA Shawe-Taylor domain parameter generation mechanism, denoted
7095 **CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN**, is a domain parameter generation mechanism
7096 based on the Digital Signature Algorithm defined in FIPS PUB 186-4, section Appendix A.1.2
7097 Construction and Validation of Provable Primes p and q.

7098 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
7099 returns the seed (*pSeed*) and the length (*ulSeedLen*).

7100 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
7101 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
7102 specified in the **CKA_SUBPRIME_BITS** attribute of the template.

7103 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
7104 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
7105 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
7106 template, or else are assigned default initial values.

7107 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7108 specify the supported range of DSA prime sizes, in bits.

7109 6.2.10 DSA base domain parameter generation

7110 The DSA base domain parameter generation mechanism, denoted **CKM_DSA_FIPS_G_GEN**, is a base
7111 parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-4,
7112 section Appendix A.2 Generation of Generator G.

7113 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash the seed
7114 (*pSeed*) and the length (*ulSeedLen*) and the index value.

7115 The mechanism generates the DSA base with the domain parameter specified in the **CKA_PRIME** and
7116 **CKA_SUBPRIME** attributes of the template.

7117 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_BASE** attributes to the new
7118 object. Other attributes supported by the DSA domain parameter types may also be specified in the
7119 template, or else are assigned default initial values.

7120 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7121 specify the supported range of DSA prime sizes, in bits.

7122 6.2.11 DSA without hashing

7123 The DSA without hashing mechanism, denoted **CKM_DSA**, is a mechanism for single-part signatures and
7124 verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism
7125 corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash
7126 value.)

7127 For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the
7128 concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

7129 It does not have a parameter.

7130 Constraints on key types and the length of data are summarized in the following table:

7131 *Table 53, DSA: Key And Data Length*

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

6.2.12 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 54, 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.

6.2.13 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

L = 2048, *N* = 224

L = 2048, *N* = 256

L = 3072, *N* = 256

6.2.14 DSA with SHA-224

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA224**, 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 SHA-224.

For the purposes of this mechanism, a DSA signature is a string of length $2 \times \text{subprime}$, 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 55, DSA with SHA-244: Key And Data Length

Function	Key type	Input length	Output length
C_Sign	DSA private key	any	$2 \times \text{subprime}$ length
C_Verify	DSA public key	any, $2 \times \text{subprime}$ 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.

6.2.15 DSA with SHA-256

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA256**, 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 SHA-256.

For the purposes of this mechanism, a DSA signature is a string of length $2 \times \text{subprime}$, 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 56, DSA with SHA-256: Key And Data Length

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

² Data length, signature length.

6.2.16 DSA with SHA-384

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA384**, 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 SHA-384.

For the purposes of this mechanism, a DSA signature is a string of length $2 \times \text{subprime}$, 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 57, 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

² Data length, signature length.

6.2.17 DSA with SHA-512

The DSA with SHA-1 mechanism, denoted **CKM_DSA_SHA512**, 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 SHA-512.

For the purposes of this mechanism, a DSA signature is a string of length 2*subprime, 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 58, 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

² Data length, signature length.

6.2.18 DSA with SHA3-224

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.

For the purposes of this mechanism, a DSA signature is a string of length 2*subprime, 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 59, 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

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.

6.2.19 DSA with SHA3-256

The DSA with SHA3-256 mechanism, denoted **CKM_DSA_SHA3_256**, 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 SHA3-256.

For the purposes of this mechanism, a DSA signature is a string of length $2 \times \text{subprime}$, 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 60, DSA with SHA3-256: Key And Data Length

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

² Data length, signature length.

6.2.20 DSA with SHA3-384

The DSA with SHA3-384 mechanism, denoted **CKM_DSA_SHA3_384**, 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 SHA3-384.

For the purposes of this mechanism, a DSA signature is a string of length $2 \times \text{subprime}$, 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 61, DSA with SHA3-384: Key And Data Length

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

² Data length, signature length.

6.2.21 DSA with SHA3-512

The DSA with SHA3-512 mechanism, denoted **CKM_DSA_SHA3_512**, 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 SHA3-512.

For the purposes of this mechanism, a DSA signature is a string of length $2 \times \text{subprime}$, 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 62, 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

² Data length, signature length.

6.3 Elliptic Curve

The Elliptic Curve (EC) cryptosystem in this document was originally based on the one described in the ANSI X9.62 and X9.63 standards developed by the ANSI X9F1 working group.

The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves were always represented in their Weierstrass form. Since that time, new curves represented in Edwards form (RFC 8032) and Montgomery form (RFC 7748) have become more common. To support these new curves, the EC cryptosystem in this document has been extended from the original. Additional key generation mechanisms have been added as well as an additional signature generation mechanism.

Table 63, 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							✓
CKM_ECMQV_DERIVE							✓

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

Table 64, 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 if the mechanism can be used with EC domain parameters of the choice curveName

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

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

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

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

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

In these specifications, an EC public key (i.e. EC point Q) or the base point G when the **ecParameters** choice is used can be represented as an octet string of the uncompressed form or the compressed form. The **CK_MECHANISM_INFO** structure **CKF_EC_UNCOMPRESS** flag identifies a Cryptoki library supporting the uncompressed form whereas the **CKF_EC_COMPRESS** flag identifies a Cryptoki library

7275 supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or
7276 both of these flags for each EC mechanism.

7277 Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation
7278 of domain parameters or one form may encounter difficulties achieving interoperability with other
7279 implementations.

7280 If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the
7281 attempt should fail with the error code CKR_CURVE_NOT_SUPPORTED. If an attempt to create,
7282 generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain
7283 parameters is made, that attempt should fail with the error code CKR_DOMAIN_PARAMS_INVALID. If
7284 an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that
7285 attempt should fail with the error code CKR_TEMPLATE_INCONSISTENT.

7286 6.3.1 EC Signatures

7287 For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at
7288 most two times $nLen$ octets, where $nLen$ is the length in octets of the base point order n . The signature
7289 octets correspond to the concatenation of the ECDSA values r and s , both represented as an octet string
7290 of equal length of at most $nLen$ with the most significant byte first. If r and s have different octet length,
7291 the shorter of both must be padded with leading zero octets such that both have the same octet length.
7292 Loosely spoken, the first half of the signature is r and the second half is s . For signatures created by a
7293 token, the resulting signature is always of length $2nLen$. For signatures passed to a token for verification,
7294 the signature may have a shorter length but must be composed as specified before.

7295 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
7296 length of n will be used. Any truncation is done by the token.

7297 Note: For applications, it is recommended to encode the signature as an octet string of length two times
7298 $nLen$ if possible. This ensures that the application works with PKCS#11 modules which have been
7299 implemented based on an older version of this document. Older versions required all signatures to have
7300 length two times $nLen$. It may be impossible to encode the signature with the maximum length of two
7301 times $nLen$ if the application just gets the integer values of r and s (i.e. without leading zeros), but does
7302 not know the base point order n , because r and s can have any value between zero and the base point
7303 order n .

7304 An EdDSA signature is an octet string of even length which is two times $nLen$ octets, where $nLen$ is
7305 calculated as EdDSA parameter b divided by 8. The signature octets correspond to the concatenation of
7306 the EdDSA values R and S as defined in [RFC 8032], both represented as an octet string of equal length
7307 of $nLen$ bytes in little endian order.

7308 6.3.2 Definitions

7309 This section defines the key types “CKK_EC”, “CKK_EC_EDWARDS” and “CKK_EC_MONTGOMERY”
7310 for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

7311 Note: CKK_ECDSA is deprecated. It is replaced by CKK_EC.

7312 Mechanisms:

7313

7314 CKM_EC_KEY_PAIR_GEN

7315 CKM_EC_EDWARDS_KEY_PAIR_GEN

7316 CKM_EC_MONTGOMERY_KEY_PAIR_GEN

7317 CKM_ECDSA

7318 CKM_ECDSA_SHA1

7319 CKM_ECDSA_SHA224

7320 CKM_ECDSA_SHA256

7321 CKM_ECDSA_SHA384

7322	CKM_ECDSA_SHA512
7323	CKM_ECDSA_SHA3_224
7324	CKM_ECDSA_SHA3_256
7325	CKM_ECDSA_SHA3_384
7326	CKM_ECDSA_SHA3_512
7327	CKM_EDDSA
7328	CKM_XEDDSA
7329	CKM_ECDH1_DERIVE
7330	CKM_ECDH1_COFACTOR_DERIVE
7331	CKM_ECMQV_DERIVE
7332	CKM_ECDH_AES_KEY_WRAP
7333	
7334	CKD_NULL
7335	CKD_SHA1_KDF
7336	CKD_SHA224_KDF
7337	CKD_SHA256_KDF
7338	CKD_SHA384_KDF
7339	CKD_SHA512_KDF
7340	CKD_SHA3_224_KDF
7341	CKD_SHA3_256_KDF
7342	CKD_SHA3_384_KDF
7343	CKD_SHA3_512_KDF
7344	CKD_SHA1_KDF_SP800
7345	CKD_SHA224_KDF_SP800
7346	CKD_SHA256_KDF_SP800
7347	CKD_SHA384_KDF_SP800
7348	CKD_SHA512_KDF_SP800
7349	CKD_SHA3_224_KDF_SP800
7350	CKD_SHA3_256_KDF_SP800
7351	CKD_SHA3_384_KDF_SP800
7352	CKD_SHA3_512_KDF_SP800
7353	CKD_BLAKE2B_160_KDF
7354	CKD_BLAKE2B_256_KDF
7355	CKD_BLAKE2B_384_KDF
7356	CKD_BLAKE2B_512_KDF

6.3.3 Short Weierstrass Elliptic Curve public key objects

Short Weierstrass EC public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC**) hold EC public keys. The following table defines the EC public key object attributes, in addition to the common attributes defined for this object class:

Table 65, 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

Refer to Table 11 for footnotes

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

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

```
Parameters ::= CHOICE {
    ecParameters      ECPParameters,
    oId                CURVES.&id({CurveNames}),
    implicitlyCA       NULL,
    curveName          PrintableString
}
```

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

The following is a sample template for creating an short Weierstrass EC public key object:

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = "An EC public key object";
CK_BYTE ecParams[] = {...};
CK_BYTE ecPoint[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
    {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

6.3.4 Short Weierstrass Elliptic Curve private key objects

Short Weierstrass EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC**) hold EC private keys. See Section 6.3 for more information about EC. The following table defines the EC private key object attributes, in addition to the common attributes defined for this object class:

Table 66, 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 <i>d</i>

Refer to Table 11 for footnotes

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

```
Parameters ::= CHOICE {
    ecParameters      ECPParameters,
    oId                CURVES.&id({CurveNames}),
    implicitlyCA       NULL,
    curveName          PrintableString
}
```

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

The following is a sample template for creating an short Weierstrass EC private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC;
CK_UTF8CHAR label[] = "An EC private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.3.5 Edwards Elliptic Curve public key objects

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

Table 67, 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	Public key bytes in little endian order as defined in RFC 8032

Refer to Table 11 for footnotes

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

```
Parameters ::= CHOICE {
    ecParameters      ECPParameters,
    oId                CURVES.&id({CurveNames}),
    implicitlyCA       NULL,
    curveName          PrintableString
}
```

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

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
CK_UTF8CHAR label[] = "An Edwards EC public key object";
CK_BYTE ecParams[] = {0x13, 0x0c, 0x65, 0x64, 0x77, 0x61,
    0x72, 0x64, 0x73, 0x32, 0x35, 0x35, 0x31, 0x39};
CK_BYTE ecPoint[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
    {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

6.3.6 Edwards Elliptic Curve private key objects

Edwards EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC_EDWARDS**) hold Edwards EC private keys. See Section 6.3 for more information about EC. The following table defines the Edwards EC private key object attributes, in addition to the common attributes defined for this object class:

Table 68, 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	Private key bytes in little endian order as defined in RFC 8032

Refer to Table 11 for footnotes

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

```
Parameters ::= CHOICE {
    ecParameters      ECPParameters,
    oid               CURVES.&id({CurveNames}),
    implicitlyCA       NULL,
    curveName          PrintableString
}
```

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

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_EC_EDWARDS;
CK_UTF8CHAR label[] = "An Edwards EC private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE ecParams[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.3.7 Montgomery Elliptic Curve public key objects

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

Table 69, 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	Public key bytes in little endian order as defined in RFC 7748

Refer to Table 11 for footnotes

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

```
Parameters ::= CHOICE {
    ecParameters      ECPParameters,
    oId                CURVES.&id({CurveNames}),
    implicitlyCA       NULL,
    curveName          PrintableString
}
```

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

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

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;
CK_UTF8CHAR label[] = "A Montgomery EC public key object";
CK_BYTE ecParams[] = {...};
CK_BYTE ecPoint[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},
    {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}
};
```

6.3.8 Montgomery Elliptic Curve private key objects

Montgomery EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC_MONTGOMERY**) hold Montgomery EC private keys. See Section 6.3 for more information about EC. The following table defines the Montgomery EC private key object attributes, in addition to the common attributes defined for this object class:

Table 70, Montgomery 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	Private key bytes in little endian order as defined in RFC 7748

Refer to Table 11 for footnotes

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

```
Parameters ::= CHOICE {  
    ecParameters      ECPParameters,  
    oId                CURVES.&id({CurveNames}),  
    implicitlyCA       NULL,  
    curveName          PrintableString  
}
```

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

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

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;  
CK_UTF8CHAR label[] = "A Montgomery EC private key object";  
CK_BYTE subject[] = {...};  
CK_BYTE id[] = {123};  
CK_BYTE ecParams[] = {...};  
CK_BYTE value[] = {...};  
CK_BBOOL true = CK_TRUE;  
CK_ATTRIBUTE template[] = {  
    {CKA_CLASS, &class, sizeof(class)},  
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
    {CKA_TOKEN, &true, sizeof(true)},  
    {CKA_LABEL, label, sizeof(label)-1},  
    {CKA_SUBJECT, subject, sizeof(subject)},  
    {CKA_ID, id, sizeof(id)},  
    {CKA_SENSITIVE, &true, sizeof(true)},  
    {CKA_DERIVE, &true, sizeof(true)},  
    {CKA_VALUE, value, sizeof(value)}  
};
```

6.3.9 Elliptic Curve key pair generation

The short Weierstrass EKey pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN, is a key pair generation mechanism that uses the method defined by the ANSI X9.62 and X9.63 standards.

The short Weierstrass EC key pair generation mechanism, denoted CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS, is a key pair generation mechanism that uses the method defined by FIPS 186-4 Appendix B.4.1.

These mechanisms do not have a parameter.

These mechanisms generate EC public/private key pairs with particular EC domain parameters, as specified in the **CKA_EC_PARAMS** attribute of the template for the public key. Note that this version of Cryptoki does not include a mechanism for generating these EC domain parameters.

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

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 ECDSA 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).

6.3.10 Edwards Elliptic Curve key pair generation

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

This mechanism does not have a parameter.

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

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

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

6.3.11 Montgomery Elliptic Curve key pair generation

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

This mechanism does not have a parameter.

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

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

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 this mechanism, the only allowed values are 255 and 448 as RFC 7748 only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

6.3.12 ECDSA without hashing

Refer section 6.3.1 for signature encoding.

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

This mechanism does not have a parameter.

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

Table 71, ECDSA without hashing: Key and Data Length

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

¹ Single-part operations only.

² Data length, signature length.

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

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 ECDSA using a field of characteristic 2 which has between 2^{200} and 2^{300} elements (inclusive), 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).

6.3.13 ECDSA with hashing

Refer to section 6.3.1 for signature encoding.

The ECDSA with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 mechanism, denoted **CKM_ECDSA [SHA1|SHA224|SHA256|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]** respectively, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing with SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512 respectively.

This mechanism does not have a parameter.

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

Table 72, ECDSA with hashing: Key and Data Length

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

² Data length, signature length.

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 ECDSA 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).

6.3.14 EdDSA

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

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

Table 73, Mapping to RFC 8032 Signature Schemes

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

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

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

Table 74, EdDSA: Key and Data Length

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

² Data length, signature length.

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

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 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 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

6.3.15 XEdDSA

The XEdDSA mechanism, denoted **CKM_XEDDSA**, is a mechanism for single-part signatures and verification for XEdDSA. This mechanism implements the XEdDSA signature scheme defined in [XEDDSA]. CKM_XEDDSA operates on CKK_EC_MONTGOMERY type EC keys, which allows these keys to be used both for signing/verification and for Diffie-Hellman style key-exchanges. This double use is necessary for the Extended Triple Diffie-Hellman where the long-term identity key is used to sign short-term keys and also contributes to the DH key-exchange.

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

Table 75, XEdDSA: Key and Data Length

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

² Data length, signature length.

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 this mechanism, the only allowed values are 255 and 448 as **[XEDDSA]** only defines curves of these two sizes. A Cryptoki implementation may support one or both of these curves and should set the *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

6.3.16 EC mechanism parameters

◆ CK_EDDSA_PARAMS, CK_EDDSA_PARAMS_PTR

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

```
typedef struct CK_EDDSA_PARAMS {
    CK_BBOOL      phFlag;
    CK_ULONG      ulContextDataLen;
    CK_BYTE_PTR   pContextData;
} CK_EDDSA_PARAMS;
```

The fields of the structure have the following meanings:

phFlag Boolean value which indicates if Prehashed variant of EdDSA should used

ulContextDataLen the length in bytes of the context data where 0 ≤ ulContextDataLen ≤ 255.

pContextData context data shared between the signer and verifier

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

◆ CK_XEDDSA_PARAMS, CK_XEDDSA_PARAMS_PTR

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

```
typedef struct CK_XEDDSA_PARAMS {
    CK_XEDDSA_HASH_TYPE hash;
} CK_XEDDSA_PARAMS;
```

The fields of the structure have the following meanings:

hash a Hash mechanism to be used by the mechanism.

CK_XEDDSA_PARAMS_PTR is a pointer to a **CK_XEDDSA_PARAMS**.

◆ CK_XEDDSA_HASH_TYPE, CK_XEDDSA_HASH_TYPE_PTR

CK_XEDDSA_HASH_TYPE is used to indicate the hash function used in XEDDSA. It is defined as follows:

```
typedef CK_ULONG CK_XEDDSA_HASH_TYPE;
```


7758 The following table lists the defined functions.

7759 Table 76, EC: Key Derivation Functions

Source Identifier
CKM_BLAKE2B_256
CKM_BLAKE2B_512
CKM_SHA3_256
CKM_SHA3_512
CKM_SHA256
CKM_SHA512

7760

7761 **CK_XEDDSA_HASH_TYPE_PTR** is a pointer to a **CK_XEDDSA_HASH_TYPE**.

7762

7763 ♦ **CK_EC_KDF_TYPE, CK_EC_KDF_TYPE_PTR**

7764 **CK_EC_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying data
7765 from a shared secret. The key derivation function will be used by the EC key agreement schemes. It is
7766 defined as follows:

7767 typedef CK_ULONG CK_EC_KDF_TYPE;

7768

7769 The following table lists the defined functions.

7770 Table 77, 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

7771 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
7772 derivation function.

7773 The key derivation functions

7774 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**, which are
7775 based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
7776 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

7777 The key derivation functions

7778 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**,
7779 which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
7780 respectively, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section
7781 5.8.1.1.

7782 The key derivation functions **CKD_BLAKE2B_[160|256|384|512]_KDF**, which are based on the Blake2b
7783 family of hashes, derive keying data from the shared secret value as defined in [FIPS SP800-56A] section
7784 5.8.1.1. **CK_EC_KDF_TYPE_PTR** is a pointer to a **CK_EC_KDF_TYPE**.

7785

7786 ♦ **CK_ECDH1_DERIVE_PARAMS, CK_ECDH1_DERIVE_PARAMS_PTR**

7787 **CK_ECDH1_DERIVE_PARAMS** is a structure that provides the parameters for the
7788 **CKM_ECDH1_DERIVE** and **CKM_ECDH1_COFACTOR_DERIVE** key derivation mechanisms, where
7789 each party contributes one key pair. The structure is defined as follows:

```
7790     typedef struct CK_ECDH1_DERIVE_PARAMS {
7791         CK_EC_KDF_TYPE    kdf;
7792         CK_ULONG           ulSharedDataLen;
7793         CK_BYTE_PTR        pSharedData;
7794         CK_ULONG           ulPublicDataLen;
7795         CK_BYTE_PTR        pPublicData;
7796     } CK_ECDH1_DERIVE_PARAMS;
```

7797

7798 The fields of the structure have the following meanings:

7799	kdf	key derivation function used on the shared secret value
7800	ulSharedDataLen	the length in bytes of the shared info
7801	pSharedData	some data shared between the two parties
7802	ulPublicDataLen	the length in bytes of the other party's EC public key

7845	hPrivateKey	key handle for second EC private key value
7846	ulPublicDataLen2	the length in bytes of the other party's second EC public key
7847	pPublicData2	pointer to other party's second EC public key value. Encoding rules
7848		are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

7849 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be
7850 zero. With the key derivation function **CKD_SHA1_KDF**, an optional *pSharedData* may be supplied,
7851 which consists of some data shared by the two parties intending to share the shared secret. Otherwise,
7852 *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

7853 **CK_ECDH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECDH2_DERIVE_PARAMS**.

7854

7855 ♦ **CK_ECMQV_DERIVE_PARAMS, CK_ECMQV_DERIVE_PARAMS_PTR**

7856 **CK_ECMQV_DERIVE_PARAMS** is a structure that provides the parameters to the
7857 **CKM_ECMQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
7858 structure is defined as follows:

```

7859     typedef struct CK_ECMQV_DERIVE_PARAMS {
7860         CK_EC_KDF_TYPE      kdf;
7861         CK_ULONG             ulSharedDataLen;
7862         CK_BYTE_PTR          pSharedData;
7863         CK_ULONG             ulPublicDataLen;
7864         CK_BYTE_PTR          pPublicData;
7865         CK_ULONG             ulPrivateKeyLen;
7866         CK_OBJECT_HANDLE     hPrivateKey;
7867         CK_ULONG             ulPublicDataLen2;
7868         CK_BYTE_PTR          pPublicData2;
7869         CK_OBJECT_HANDLE     publicKey;
7870     } CK_ECMQV_DERIVE_PARAMS;

```

7871

7872 The fields of the structure have the following meanings:

7873	kdf	key derivation function used on the shared secret value
7874	ulSharedDataLen	the length in bytes of the shared info
7875	pSharedData	some data shared between the two parties
7876	ulPublicDataLen	the length in bytes of the other party's first EC public key
7877	pPublicData	pointer to other party's first EC public key value. Encoding rules are
7878		as per pPublicData of CK_ECDH1_DERIVE_PARAMS
7879	ulPrivateKeyLen	the length in bytes of the second EC private key
7880	hPrivateKey	key handle for second EC private key value
7881	ulPublicDataLen2	the length in bytes of the other party's second EC public key
7882	pPublicData2	pointer to other party's second EC public key value. Encoding rules
7883		are as per pPublicData of CK_ECDH1_DERIVE_PARAMS
7884	publicKey	Handle to the first party's ephemeral public key

7885 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be
7886 zero. With the key derivation functions
7887 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**,
7888 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**, an

optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

CK_ECMQV_DERIVE_PARAMS_PTR is a pointer to a **CK_ECMQV_DERIVE_PARAMS**.

6.3.17 Elliptic Curve Diffie-Hellman key derivation

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 for short Weierstrass EC keys and RFC 7748 for Montgomery keys, where each party contributes one key pair all using the same EC domain parameters.

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:

Table 78: ECDH: Allowed Key Types

Function	Key type
C_Derive	CKK_EC or CKK_EC_MONTGOMERY

6.3.18 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. Co-factor 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:

Table 79: ECDH with cofactor: Allowed Key Types

Function	Key type
C_Derive	CKK_EC

6.3.19 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 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:

Table 80: ECDH MQV: Allowed Key Types

Function	Key type
C_Derive	CKK_EC

6.3.20 ECDH AES KEY WRAP

The ECDH AES KEY WRAP mechanism, denoted **CKM_ECDH_AES_KEY_WRAP**, is a mechanism based on Elliptic Curve public-key crypto-system and the AES key wrap mechanism. It supports single-part key wrapping; and key unwrapping.

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

The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC key.

- A temporary AES key is derived from a temporary EC key and the wrapping EC key using the **CKM_ECDH1_DERIVE** mechanism.
- The derived AES key is used for wrapping the target key using the **CKM_AES_KEY_WRAP_KWP** mechanism.

For wrapping, the mechanism -

- Generates a temporary random EC key (transport key) having the same parameters as the wrapping EC key (and domain parameters). Saves the transport key public key material.
- Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private key of the transport EC key and the public key of wrapping EC key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.
- Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- Zeroizes the temporary AES key and EC transport private key.
- Concatenates public key material of the transport key and output the concatenated blob. The first part is the public key material of the transport key and the second part is the wrapped target key.

The private target key will be encoded as defined in section 6.7.

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

For unwrapping, the mechanism -

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

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

- Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen and pSharedData using the private part of unwrapping EC key and the public part of the transport EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.

- Un-wraps the target key from the second part with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- Zeroizes the temporary AES key.

Table 81, 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						✓	
¹ SR = SignRecover, VR = VerifyRecover							

Constraints on key types are summarized in the following table:

Table 82: ECDH AES Key Wrap: Allowed Key Types

Function	Key type
C_Wrap / C_Unwrap	CKK_EC or CKK_EC_MONTGOMERY

6.3.21 ECDH AES KEY WRAP mechanism parameters

◆ CK_ECDH_AES_KEY_WRAP_PARAMS; CK_ECDH_AES_KEY_WRAP_PARAMS_PTR

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

```
typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
    CK_ULONG          ulAESKeyBits;
    CK_EC_KDF_TYPE    kdf;
    CK_ULONG          ulSharedDataLen;
    CK_BYTE_PTR       pSharedData;
} CK_ECDH_AES_KEY_WRAP_PARAMS;
```

The fields of the structure have the following meanings:

ulAESKeyBits	length of the temporary AES key in bits. Can be only 128, 192 or 256.
kdf	key derivation function used on the shared secret value to generate AES key.
ulSharedDataLen	the length in bytes of the shared info
pSharedData	Some data shared between the two parties

CK_ECDH_AES_KEY_WRAP_PARAMS_PTR is a pointer to a **CK_ECDH_AES_KEY_WRAP_PARAMS**.

6.3.22 FIPS 186-4

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

P-192, P-224, P-256, P-384, P-521

K-163, B-163, K-233, B-233

K-283, B-283, K-409, B-409

K-571, B-571

6.4 Diffie-Hellman

Table 83, Diffie-Hellman Mechanisms vs. Functions

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

6.4.1 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_PARAMETER_GEN

CKM_DH_PKCS_DERIVE

CKM_X9_42_DH_KEY_PAIR_GEN

CKM_X9_42_DH_PARAMETER_GEN

CKM_X9_42_DH_DERIVE

CKM_X9_42_DH_HYBRID_DERIVE

CKM_X9_42_MQV_DERIVE

6.4.2 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 84, 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 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:

```
CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman public key object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.4.3 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 85, 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

Refer to Table 11 for footnotes

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

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

```

CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key
    object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_VALUE, value, sizeof(value)}
};

```

6.4.4 Diffie-Hellman private key objects

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

Table 86, 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

Refer to 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.

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

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

```

CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};

```

```

8157 CK_BYTE value[] = {...};
8158 CK_BBOOL true = CK_TRUE;
8159 CK_ATTRIBUTE template[] = {
8160     {CKA_CLASS, &class, sizeof(class)},
8161     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8162     {CKA_TOKEN, &true, sizeof(true)},
8163     {CKA_LABEL, label, sizeof(label)-1},
8164     {CKA_SUBJECT, subject, sizeof(subject)},
8165     {CKA_ID, id, sizeof(id)},
8166     {CKA_SENSITIVE, &true, sizeof(true)},
8167     {CKA_DERIVE, &true, sizeof(true)},
8168     {CKA_PRIME, prime, sizeof(prime)},
8169     {CKA_BASE, base, sizeof(base)},
8170     {CKA_VALUE, value, sizeof(value)}
8171 };

```

8172 6.4.5 X9.42 Diffie-Hellman private key objects

8173 X9.42 Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_X9_42_DH**)
8174 hold X9.42 Diffie-Hellman private keys. The following table defines the X9.42 Diffie-Hellman private key
8175 object attributes, in addition to the common attributes defined for this object class:

8176 Table 87, 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

8177 Refer to Table 11 for footnotes

8178 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the “X9.42 Diffie-
8179 Hellman domain parameters”. Depending on the token, there may be limits on the length of the key
8180 components. See the ANSI X9.42 standard for more information on X9.42 Diffie-Hellman keys.

8181 Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain
8182 parameters are *not* specified in the key’s template. This is because X9.42 Diffie-Hellman private keys are
8183 only generated as part of a X9.42 Diffie-Hellman key *pair*, and the X9.42 Diffie-Hellman domain
8184 parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

8185 The following is a sample template for creating a X9.42 Diffie-Hellman private key object:

```

8186 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
8187 CK_KEY_TYPE keyType = CKK_X9_42_DH;
8188 CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman private key object";
8189 CK_BYTE subject[] = {...};
8190 CK_BYTE id[] = {123};
8191 CK_BYTE prime[] = {...};
8192 CK_BYTE base[] = {...};
8193 CK_BYTE subprime[] = {...};
8194 CK_BYTE value[] = {...};
8195 CK_BBOOL true = CK_TRUE;
8196 CK_ATTRIBUTE template[] = {
8197     {CKA_CLASS, &class, sizeof(class)},

```

```

8198     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8199     {CKA_TOKEN, &true, sizeof(true)},
8200     {CKA_LABEL, label, sizeof(label)-1},
8201     {CKA_SUBJECT, subject, sizeof(subject)},
8202     {CKA_ID, id, sizeof(id)},
8203     {CKA_SENSITIVE, &true, sizeof(true)},
8204     {CKA_DERIVE, &true, sizeof(true)},
8205     {CKA_PRIME, prime, sizeof(prime)},
8206     {CKA_BASE, base, sizeof(base)},
8207     {CKA_SUBPRIME, subprime, sizeof(subprime)},
8208     {CKA_VALUE, value, sizeof(value)}
8209 };

```

8210 6.4.6 Diffie-Hellman domain parameter objects

8211 Diffie-Hellman domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
8212 **CKK_DH**) hold Diffie-Hellman domain parameters. The following table defines the Diffie-Hellman domain
8213 parameter object attributes, in addition to the common attributes defined for this object class:

8214 *Table 88, 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.

8215 Refer to Table 11 for footnotes

8216 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
8217 parameters”. Depending on the token, there may be limits on the length of the key components. See
8218 PKCS #3 for more information on Diffie-Hellman domain parameters.

8219 The following is a sample template for creating a Diffie-Hellman domain parameter object:

```

8220 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
8221 CK_KEY_TYPE keyType = CKK_DH;
8222 CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters
8223     object";
8224 CK_BYTE prime[] = {...};
8225 CK_BYTE base[] = {...};
8226 CK_BBOOL true = CK_TRUE;
8227 CK_ATTRIBUTE template[] = {
8228     {CKA_CLASS, &class, sizeof(class)},
8229     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8230     {CKA_TOKEN, &true, sizeof(true)},
8231     {CKA_LABEL, label, sizeof(label)-1},
8232     {CKA_PRIME, prime, sizeof(prime)},
8233     {CKA_BASE, base, sizeof(base)},
8234 };

```

8235 6.4.7 X9.42 Diffie-Hellman domain parameters objects

8236 X9.42 Diffie-Hellman domain parameters objects (object class **CKO_DOMAIN_PARAMETERS**, key type
8237 **CKK_X9_42_DH**) hold X9.42 Diffie-Hellman domain parameters. The following table defines the X9.42

Diffie-Hellman domain parameters object attributes, in addition to the common attributes defined for this object class:

Table 89, 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.

Refer to Table 11 for footnotes

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

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

```
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain
    parameters object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
CK_BYTE subprime[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
};
```

6.4.8 PKCS #3 Diffie-Hellman key pair generation

The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted **CKM_DH_PKCS_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3. This is what PKCS #3 calls “phase I”. It does not have a parameter.

The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as specified in the **CKA_PRIME** and **CKA_BASE** attributes of the template for the public key. If the **CKA_VALUE_BITS** attribute of the private key is specified, the mechanism limits the length in bits of the private value, as described in PKCS #3.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and **CKA_VALUE** (and the **CKA_VALUE_BITS** attribute, if it is not already provided in the template) attributes to the new private key; other attributes required by the Diffie-Hellman public and private key types must be specified in the templates.

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

6.4.9 PKCS #3 Diffie-Hellman domain parameter generation

The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted **CKM_DH_PKCS_PARAMETER_GEN**, is a domain parameter generation mechanism based on Diffie-Hellman key agreement, as defined in PKCS #3.

It does not have a parameter.

The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template.

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

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

6.4.10 PKCS #3 Diffie-Hellman key derivation

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

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

This mechanism derives a secret key from a Diffie-Hellman private key and the public value of the other party. It computes a Diffie-Hellman secret value from the public value and private key according to PKCS #3, 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 extractability²:

- 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 supported range of Diffie-Hellman prime sizes, in bits.

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**.

6.4.11 X9.42 Diffie-Hellman mechanism parameters

◆ CK_X9_42_DH_KDF_TYPE, CK_X9_42_DH_KDF_TYPE_PTR

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

```
typedef CK_ULONG CK_X9_42_DH_KDF_TYPE;
```

The following table lists the defined functions.

Table 90, X9.42 Diffie-Hellman Key Derivation Functions

Source Identifier
CKD_NULL
CKD_SHA1_KDF_ASN1
CKD_SHA1_KDF_CONCATENATE

The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key derivation function whereas the key derivation functions **CKD_SHA1_KDF_ASN1** and **CKD_SHA1_KDF_CONCATENATE**, which are both based on SHA-1, derive keying data from the shared secret value as defined in the ANSI X9.42 standard.

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

◆ CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR

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

```
typedef struct CK_X9_42_DH1_DERIVE_PARAMS {  
    CK_X9_42_DH_KDF_TYPE    kdf;  
    CK_ULONG                 ulOtherInfoLen;  
    CK_BYTE_PTR              pOtherInfo;  
    CK_ULONG                 ulPublicDataLen;  
    CK_BYTE_PTR              pPublicData;  
} CK_X9_42_DH1_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

kdf	key derivation function used on the shared secret value
ulOtherInfoLen	the length in bytes of the other info
pOtherInfo	some data shared between the two parties
ulPublicDataLen	the length in bytes of the other party's X9.42 Diffie-Hellman public key
pPublicData	pointer to other party's X9.42 Diffie-Hellman public key value

With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero. With the key derivation function **CKD_SHA1_KDF_ASN1**, *pOtherInfo* must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some

8356 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
8357 NULL and *ulOtherInfoLen* must be zero.

8358 **CK_X9_42_DH1_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH1_DERIVE_PARAMS**.

8359 • **CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR**

8360 **CK_X9_42_DH2_DERIVE_PARAMS** is a structure that provides the parameters to the
8361 **CKM_X9_42_DH_HYBRID_DERIVE** and **CKM_X9_42_MQV_DERIVE** key derivation mechanisms,
8362 where each party contributes two key pairs. The structure is defined as follows:

```
8363     typedef struct CK_X9_42_DH2_DERIVE_PARAMS {  
8364         CK_X9_42_DH_KDF_TYPE      kdf;  
8365         CK_ULONG                   ulOtherInfoLen;  
8366         CK_BYTE_PTR                pOtherInfo;  
8367         CK_ULONG                   ulPublicDataLen;  
8368         CK_BYTE_PTR                pPublicData;  
8369         CK_ULONG                   ulPrivateDataLen;  
8370         CK_OBJECT_HANDLE           hPrivateData;  
8371         CK_ULONG                   ulPublicDataLen2;  
8372         CK_BYTE_PTR                pPublicData2;  
8373     } CK_X9_42_DH2_DERIVE_PARAMS;
```

8374

8375 The fields of the structure have the following meanings:

8376	kdf	key derivation function used on the shared secret value
8377	ulOtherInfoLen	the length in bytes of the other info
8378	pOtherInfo	some data shared between the two parties
8379	ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman
8380		public key
8381	pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
8382	ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
8383	hPrivateData	key handle for second X9.42 Diffie-Hellman private key value
8384	ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman
8385		public key
8386	pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key
8387		value

8388 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
8389 With the key derivation function **CKD_SHA1_KDF_ASN1**, *pOtherInfo* must be supplied, which contains
8390 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
8391 the two parties intending to share the shared secret. With the key derivation function
8392 **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
8393 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be
8394 NULL and *ulOtherInfoLen* must be zero.

8395 **CK_X9_42_DH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH2_DERIVE_PARAMS**.

• **CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR**

CK_X9_42_MQV_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_X9_42_MQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The structure is defined as follows:

```
typedef struct CK_X9_42_MQV_DERIVE_PARAMS {
    CK_X9_42_DH_KDF_TYPE    kdf;
    CK_ULONG                 ulOtherInfoLen;
    CK_BYTE_PTR              pOtherInfo;
    CK_ULONG                 ulPublicDataLen;
    CK_BYTE_PTR              pPublicData;
    CK_ULONG                 ulPrivateDataLen;
    CK_OBJECT_HANDLE         hPrivateData;
    CK_ULONG                 ulPublicDataLen2;
    CK_BYTE_PTR              pPublicData2;
    CK_OBJECT_HANDLE         publicKey;
} CK_X9_42_MQV_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

kdf	key derivation function used on the shared secret value
ulOtherInfoLen	the length in bytes of the other info
pOtherInfo	some data shared between the two parties
ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman public key
pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
hPrivateData	key handle for second X9.42 Diffie-Hellman private key value
ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman public key
pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key value
publicKey	Handle to the first party's ephemeral public key

With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero. With the key derivation function **CKD_SHA1_KDF_ASN1**, *pOtherInfo* must be supplied, which contains an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by the two parties intending to share the shared secret. With the key derivation function **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.

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

6.4.12 X9.42 Diffie-Hellman key pair generation

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

It does not have a parameter.

8440 The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and
8441 subprime, as specified in the **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attributes of the template
8442 for the public key.

8443 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
8444 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, **CKA_SUBPRIME**, and
8445 **CKA_VALUE** attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman
8446 public and private key types must be specified in the templates.

8447 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8448 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8449 6.4.13 X9.42 Diffie-Hellman domain parameter generation

8450 The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted
8451 **CKM_X9_42_DH_PARAMETER_GEN**, is a domain parameters generation mechanism based on X9.42
8452 Diffie-Hellman key agreement, as defined in the ANSI X9.42 standard.

8453 It does not have a parameter.

8454 The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime
8455 length in bits, as specified in the **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes of the
8456 template for the domain parameters.

8457 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**,
8458 **CKA_SUBPRIME**, **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes to the new object. Other
8459 attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the
8460 template for the domain parameters, or else are assigned default initial values.

8461 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8462 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits.

8463 6.4.14 X9.42 Diffie-Hellman key derivation

8464 The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM_X9_42_DH_DERIVE**, is a
8465 mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the
8466 ANSI X9.42 standard, where each party contributes one key pair, all using the same X9.42 Diffie-Hellman
8467 domain parameters.

8468 It has a parameter, a **CK_X9_42_DH1_DERIVE_PARAMS** structure.

8469 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
8470 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
8471 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
8472 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
8473 type must be specified in the template. Note that in order to validate this mechanism it may be required to
8474 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
8475 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

8476 This mechanism has the following rules about key sensitivity and extractability:

- 8477 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
8478 be specified to be either **CK_TRUE** or **CK_FALSE**. If omitted, these attributes each take on some
8479 default value.
- 8480 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_FALSE**, then the derived key
8481 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_TRUE**, then the
8482 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8483 **CKA_SENSITIVE** attribute.
- 8484 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to **CK_FALSE**, then the
8485 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8486 **CK_TRUE**, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8487 value from its **CKA_EXTRACTABLE** attribute.

8488 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8489 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8490 6.4.15 X9.42 Diffie-Hellman hybrid key derivation

8491 The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted
8492 **CKM_X9_42_DH_HYBRID_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman
8493 hybrid key agreement scheme, as defined in the ANSI X9.42 standard, where each party contributes two
8494 key pair, all using the same X9.42 Diffie-Hellman domain parameters.

8495 It has a parameter, a **CK_X9_42_DH2_DERIVE_PARAMS** structure.

8496 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
8497 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
8498 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
8499 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
8500 type must be specified in the template. Note that in order to validate this mechanism it may be required to
8501 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
8502 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

8503 This mechanism has the following rules about key sensitivity and extractability:

- 8504 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
8505 be specified to be either **CK_TRUE** or **CK_FALSE**. If omitted, these attributes each take on some
8506 default value.
- 8507 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_FALSE**, then the derived key
8508 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_TRUE**, then the
8509 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8510 **CKA_SENSITIVE** attribute.
- 8511 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to **CK_FALSE**, then the
8512 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8513 **CK_TRUE**, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8514 value from its **CKA_EXTRACTABLE** attribute.

8515 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8516 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8517 6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

8518 The X9.42 Diffie-Hellman Menezes-Qu-Vanstone (MQV) key derivation mechanism, denoted
8519 **CKM_X9_42_MQV_DERIVE**, is a mechanism for key derivation based the MQV scheme, as defined in
8520 the ANSI X9.42 standard, where each party contributes two key pairs, all using the same X9.42 Diffie-
8521 Hellman domain parameters.

8522 It has a parameter, a **CK_X9_42_MQV_DERIVE_PARAMS** structure.

8523 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
8524 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
8525 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
8526 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
8527 type must be specified in the template. Note that in order to validate this mechanism it may be required to
8528 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
8529 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

8530 This mechanism has the following rules about key sensitivity and extractability:

- 8531 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
8532 be specified to be either **CK_TRUE** or **CK_FALSE**. If omitted, these attributes each take on some
8533 default value.
- 8534 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_FALSE**, then the derived key
8535 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_TRUE**, then the

8536 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
8537 **CKA_SENSITIVE** attribute.

- 8538 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
8539 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
8540 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
8541 value from its **CKA_EXTRACTABLE** attribute.

8542 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8543 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

8544 6.5 Extended Triple Diffie-Hellman (x3dh)

8545 The Extended Triple Diffie-Hellman mechanism described here is the one described in
8546 [SIGNAL].

8547

8548 *Table 91, Extended Triple Diffie-Hellman Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwr ap	Derive
CKM_X3DH_INITIALIZE							✓
CKM_X3DH_RESPOND							✓

8549 6.5.1 Definitions

8550 Mechanisms:

8551 CKM_X3DH_INITIALIZE

8552 CKM_X3DH_RESPOND

8553 6.5.2 Extended Triple Diffie-Hellman key objects

8554 Extended Triple Diffie-Hellman uses Elliptic Curve keys in Montgomery representation
8555 (**CKK_EC_MONTGOMERY**). Three different kinds of keys are used, they differ in their lifespan:

- 8556 • identity keys are long-term keys, which identify the peer,
- 8557 • prekeys are short-term keys, which should be rotated often (weekly to hourly)
- 8558 • onetime prekeys are keys, which should be used only once.

8559 Any peer intending to be contacted using X3DH must publish their so-called prekey-bundle, consisting of
8560 their:

- 8561 • public Identity key,
- 8562 • current prekey, signed using XEDDSA with their identity key
- 8563 • optionally a batch of One-time public keys.

8564 6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange

8565 Initiating an Extended Triple Diffie-Hellman key exchange starts by retrieving the following required public
8566 keys (the so-called prekey-bundle) of the other peer: the Identity key, the signed public Prekey, and
8567 optionally one One-time public key.

8568 When the necessary key material is available, the initiating party calls CKM_X3DH_INITIALIZE, also
8569 providing the following additional parameters:

- 8570 • the initiators identity key

- the initiators ephemeral key (a fresh, one-time **CKK_EC_MONTGOMERY** type key)

CK_X3DH_INITIATE_PARAMS is a structure that provides the parameters to the **CKM_X3DH_INITIALIZE** key exchange mechanism. The structure is defined as follows:

```
typedef struct CK_X3DH_INITIATE_PARAMS {
    CK_X3DH_KDF_TYPE    kdf;
    CK_OBJECT_HANDLE    pPeer_identity;
    CK_OBJECT_HANDLE    pPeer_prekey;
    CK_BYTE_PTR         pPrekey_signature;
    CK_BYTE_PTR         pOnetime_key;
    CK_OBJECT_HANDLE    pOwn_identity;
    CK_OBJECT_HANDLE    pOwn_ephemeral;
} CK_X3DH_INITIATE_PARAMS;
```

Table 92, Extended Triple Diffie-Hellman Initiate Message parameters:

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	Key derivation function
pPeer_identity	Key handle	Peers public Identity key (from the prekey-bundle)
pPeer_prekey	Key Handle	Peers public prekey (from the prekey-bundle)
pPrekey_signature	Byte array	XEDDSA signature of PEER_PREKEY (from prekey-bundle)
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

6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange

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

```
typedef struct CK_X3DH_RESPOND_PARAMS {
    CK_X3DH_KDF_TYPE    kdf;
    CK_BYTE_PTR         pIdentity_id;
    CK_BYTE_PTR         pPrekey_id;
    CK_BYTE_PTR         pOnetime_id;
    CK_OBJECT_HANDLE    pInitiator_identity;
    CK_BYTE_PTR         pInitiator_ephemeral;
} CK_X3DH_RESPOND_PARAMS;
```

Table 93, Extended Triple Diffie-Hellman 1st Message parameters:

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	Key derivation function
pIdentity_id	Byte array	Peers public Identity key identifier (from the prekey-bundle)
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

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

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

- 1 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.
- 2 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.
- 3 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.

6.5.5 Extended Triple Diffie-Hellman parameters

• CK_X3DH_KDF_TYPE, CK_X3DH_KDF_TYPE_PTR

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

```
typedef CK_ULONG CK_X3DH_KDF_TYPE;
```

The following table lists the defined functions.

Table 94, X3DH: Key Derivation Functions

Source Identifier
CKD_NULL
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_512_KDF
CKD_SHA3_256_KDF

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.

CKD_SHA256_KDF
CKD_SHA3_512_KDF
CKD_SHA512_KDF

6.6 Double Ratchet

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

Table 95, Double Ratchet Mechanisms vs. Functions

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	✓					✓	

6.6.1 Definitions

This section defines the key type “CKK_X2RATCHET” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_X2RATCHET_INITIALIZE
CKM_X2RATCHET_RESPOND
CKM_X2RATCHET_ENCRYPT
CKM_X2RATCHET_DECRYPT

6.6.2 Double Ratchet secret key objects

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

Table 96, 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_NHKS	Byte array	Next Sender Header Key
CKA_X2RATCHET_NHKR	Byte array	Next Receiver Header Key
CKA_X2RATCHET_CKS	Byte array	Sender Chain key

Attribute	Data type	Meaning
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

6.6.3 Double Ratchet key derivation

The Double Ratchet key derivation mechanisms depend on who is the initiating party, and who the receiving, denoted **CKM_X2RATCHET_INITIALIZE** and **CKM_X2RATCHET_RESPOND**, are the key derivation mechanisms for the Double Ratchet. Usually the keys are derived from a shared secret by executing a X3DH key exchange.

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

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

- **CK_X2RATCHET_INITIALIZE_PARAMS;**
CK_X2RATCHET_INITIALIZE_PARAMS_PTR

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

```
typedef struct CK_X2RATCHET_INITIALIZE_PARAMS {
    CK_BYTE_PTR          sk;
    CK_OBJECT_HANDLE     peer_public_prekey;
    CK_OBJECT_HANDLE     peer_public_identity;
    CK_OBJECT_HANDLE     own_public_identity;
    CK_BBOOL             bEncryptedHeader;
    CK_ULONG             eCurve;
    CK_MECHANISM_TYPE     aeadMechanism;
    CK_X2RATCHET_KDF_TYPE kdfMechanism;
} CK_X2RATCHET_INITIALIZE_PARAMS;
```

The fields of the structure have the following meanings:

<i>sk</i>	<i>the shared secret with peer (derived using X3DH)</i>
<i>peers_public_prekey</i>	<i>Peers public prekey which the Initiator used in the X3DH</i>
<i>peers_public_identity</i>	<i>Peers public identity which the Initiator used in the X3DH</i>

8679	<i>own_public_identity</i>	<i>Initiators public identity as used in the X3DH</i>
8680	<i>bEncryptedHeader</i>	<i>whether the headers are encrypted</i>
8681	<i>eCurve</i>	<i>255 for curve 25519 or 448 for curve 448</i>
8682	<i>aeadMechanism</i>	<i>a mechanism supporting AEAD encryption</i>
8683	<i>kdfMechanism</i>	<i>a Key Derivation Mechanism, such as</i>
8684		<i>CKD_BLAKE2B_512_KDF</i>
8685	• CK_X2RATCHET_RESPOND_PARAMS;	
8686	CK_X2RATCHET_RESPOND_PARAMS_PTR	
8687	CK_X2RATCHET_RESPOND_PARAMS provides the parameters to the	
8688	CKM_X2RATCHET_RESPOND mechanism. It is defined as follows:	
8689	typedef struct CK_X2RATCHET_RESPOND_PARAMS {	
8690	CK_BYTE_PTR	sk;
8691	CK_OBJECT_HANDLE	own_prekey;
8692	CK_OBJECT_HANDLE	initiator_identity;
8693	CK_OBJECT_HANDLE	own_public_identity;
8694	CK_BBOOL	bEncryptedHeader;
8695	CK_ULONG	eCurve;
8696	CK_MECHANISM_TYPE	aeadMechanism;
8697	CK_X2RATCHET_KDF_TYPE	kdfMechanism;
8698	}	CK_X2RATCHET_RESPOND_PARAMS;
8699		
8700	The fields of the structure have the following meanings:	
8701	<i>sk</i>	<i>shared secret with the Initiator</i>
8702	<i>own_prekey</i>	<i>Own Prekey pair that the Initiator used</i>
8703	<i>initiator_identity</i>	<i>Initiators public identity key used</i>
8704	<i>own_public_identity</i>	<i>as used in the prekey bundle by the initiator in the X3DH</i>
8705	<i>bEncryptedHeader</i>	<i>whether the headers are encrypted</i>
8706	<i>eCurve</i>	<i>255 for curve 25519 or 448 for curve 448</i>
8707	<i>aeadMechanism</i>	<i>a mechanism supporting AEAD encryption</i>
8708	<i>kdfMechanism</i>	<i>a Key Derivation Mechanism, such as</i>
8709		<i>CKD_BLAKE2B_512_KDF</i>

8710 6.6.4 Double Ratchet Encryption mechanism

8711 The Double Ratchet encryption mechanism, denoted **CKM_X2RATCHET_ENCRYPT** and
8712 **CKM_X2RATCHET_DECRYPT**, are a mechanisms for single part encryption and decryption based on
8713 the Double Ratchet and its underlying AEAD cipher.

6.6.5 Double Ratchet parameters

- **CK_X2RATCHET_KDF_TYPE, CK_X2RATCHET_KDF_TYPE_PTR**

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

```
typedef CK_ULONG CK_X2RATCHET_KDF_TYPE;
```

The following table lists the defined functions.

Table 97, 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

6.7 Wrapping/unwrapping private keys

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

```
rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }

dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }

dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) ansi-x942(10046) number-type(2) 1 }

id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2)
    us(840) ansi-x9-62(10045) publicKeyType(2) 1 }

id-dsa OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) us(840) x9-57(10040) x9cm(4) 1 }

where
pkcs-1 OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }

pkcs-3 OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }
```

8750 These parameters for the algorithm identifiers have the
8751 following types, respectively:

8752 NULL

8753

```
8754 DHParameter ::= SEQUENCE {  
8755     prime          INTEGER,  -- p  
8756     base           INTEGER,  -- g  
8757     privateValueLength  INTEGER OPTIONAL  
8758 }
```

8759

```
8760 DomainParameters ::= SEQUENCE {  
8761     prime          INTEGER,  -- p  
8762     base           INTEGER,  -- g  
8763     subprime       INTEGER,  -- q  
8764     cofactor       INTEGER OPTIONAL,  -- j  
8765     validationParms ValidationParms OPTIONAL  
8766 }
```

8767

```
8768 ValidationParms ::= SEQUENCE {  
8769     Seed           BIT STRING,  -- seed  
8770     PGenCounter    INTEGER      -- parameter verification  
8771 }
```

8772

```
8773 Parameters ::= CHOICE {  
8774     ecParameters    ECParameters,  
8775     namedCurve      CURVES.&id({CurveNames}),  
8776     implicitlyCA     NULL  
8777 }
```

8778

```
8779 Dss-Parms ::= SEQUENCE {  
8780     p INTEGER,  
8781     q INTEGER,  
8782     g INTEGER  
8783 }
```

8784

8785 For the X9.42 Diffie-Hellman domain parameters, the **cofactor** and the **validationParms** optional fields
8786 should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values
8787 are not stored within the token.

8788 For the EC domain parameters, the use of **namedCurve** is recommended over the choice
8789 **ecParameters**. The choice **implicitlyCA** must not be used in Cryptoki.

8790 Within the PrivateKeyInfo type:

- 8791 • RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type
8792 requires values to be present for *all* the attributes specific to Cryptoki's RSA private key objects. In
8793 other words, if a Cryptoki library does not have values for an RSA private key's **CKA_MODULUS**,
8794 **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**,
8795 **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, and **CKA_COEFFICIENT** values, it must not create an
8796 RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.
- 8797 • Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.

- 8798 • X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
- 8799 • Short Weierstrass EC private keys are BER-encoded according to SECG SEC 1 ECPrivateKey
- 8800 ASN.1 type:

```

8801 ECPrivateKey ::= SEQUENCE {
8802     Version          INTEGER { ecPrivkeyVer1(1) }
8803     (ecPrivkeyVer1),
8804     privateKey       OCTET STRING,
8805     parameters       [0] Parameters OPTIONAL,
8806     publicKey        [1] BIT STRING OPTIONAL
8807 }

```

8808

8809 Since the EC domain parameters are placed in the PKCS #8's privateKeyAlgorithm field, the optional

8810 **parameters** field in an ECPrivateKey must be omitted. A Cryptoki application must be able to

8811 unwrap an ECPrivateKey that contains the optional **publicKey** field; however, what is done with this

8812 **publicKey** field is outside the scope of Cryptoki.

- 8813 • DSA private keys are represented as BER-encoded ASN.1 type INTEGER.

8814 Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is

8815 encrypted with the secret key. This encryption is defined in the section for the respective key wrapping

8816 mechanism.

8817 Unwrapping a wrapped private key undoes the above procedure. The ciphertext is decrypted as defined

8818 for the respective key unwrapping mechanism. The data thereby obtained are parsed as a

8819 PrivateKeyInfo type. An error will result if the original wrapped key does not decrypt properly, or if the

8820 decrypted data does not parse properly, or its type does not match the key type specified in the template

8821 for the new key. The unwrapping mechanism contributes only those attributes specified in the

8822 PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will

8823 take their default values.

8824 Earlier drafts of PKCS #11 Version 2.0 and Version 2.01 used the object identifier

```

8825 DSA OBJECT IDENTIFIER ::= { algorithm 12 }
8826 algorithm OBJECT IDENTIFIER ::= {
8827     iso(1) identifier-organization(3) oiw(14) secsig(3)
8828     algorithm(2) }

```

8829

8830 with associated parameters

```

8831 DSAParameters ::= SEQUENCE {
8832     prime1 INTEGER, -- modulus p
8833     prime2 INTEGER, -- modulus q
8834     base INTEGER -- base g
8835 }

```

8836

8837 for wrapping DSA private keys. Note that although the two structures for holding DSA domain

8838 parameters appear identical when instances of them are encoded, the two corresponding object

8839 identifiers are different.

8840 6.8 Generic secret key

8841 *Table 98, 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					✓		

6.8.1 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

6.8.2 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 99, 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 Table 11 for footnotes

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

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
CK_UTF8CHAR label[] = "A generic secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

```

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.

6.8.3 Generic secret key generation

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

It does not have a parameter.

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

```
CK_OBJECT_CLASS = CKO_SECRET_KEY;  
CK_KEY_TYPE = CKK_GENERIC_SECRET;
```

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

6.9 HMAC mechanisms

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

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

6.9.1 General block cipher mechanism parameters

• **CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR**

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

```
typedef CK_ULONG CK_MAC_GENERAL_PARAMS;
```

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

6.10 AES

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

Table 100, 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		✓					

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

6.10.1 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:

CKM_AES_KEY_GEN
CKM_AES_ECB
CKM_AES_CBC
CKM_AES_MAC
CKM_AES_MAC_GENERAL
CKM_AES_CBC_PAD
CKM_AES_OFB
CKM_AES_CFB64
CKM_AES_CFB8
CKM_AES_CFB128
CKM_AES_CFB1
CKM_AES_XCBC_MAC
CKM_AES_XCBC_MAC_96

6.10.2 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:

Table 101, 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 Table 11 for footnotes

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

```

8927     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
8928     CK_KEY_TYPE keyType = CKK_AES;
8929     CK_UTF8CHAR label[] = "An AES secret key object";
8930     CK_BYTE value[] = {...};
8931     CK_BBOOL true = CK_TRUE;
8932     CK_ATTRIBUTE template[] = {
8933         {CKA_CLASS, &class, sizeof(class)},
8934         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8935         {CKA_TOKEN, &true, sizeof(true)},
8936         {CKA_LABEL, label, sizeof(label)-1},
8937         {CKA_ENCRYPT, &true, sizeof(true)},
8938         {CKA_VALUE, value, sizeof(value)}
8939     };

```

8940

8941 **CKA_CHECK_VALUE**: The value of this attribute is derived from the key object by taking the first three
8942 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
8943 the key type of the secret key object.

8944 6.10.3 AES key generation

8945 The AES key generation mechanism, denoted **CKM_AES_KEY_GEN**, is a key generation mechanism for
8946 NIST's Advanced Encryption Standard.

8947 It does not have a parameter.

8948 The mechanism generates AES keys with a particular length in bytes, as specified in the
8949 **CKA_VALUE_LEN** attribute of the template for the key.

8950 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
8951 key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the
8952 key supports) may be specified in the template for the key, or else are assigned default initial values.

8953 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8954 specify the supported range of AES key sizes, in bytes.

8955 6.10.4 AES-ECB

8956 AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and
8957 decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and
8958 electronic codebook mode.

8959 It does not have a parameter.

8960 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
8961 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
8962 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
8963 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
8964 length as the padded input data. It does not wrap the key type, key length, or any other information about
8965 the key; the application must convey these separately.

8966 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
8967 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
8968 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
8969 attribute of the new key; other attributes required by the key type must be specified in the template.

8970 Constraints on key types and the length of data are summarized in the following table:

8971 *Table 102, 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	

8972 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8973 specify the supported range of AES key sizes, in bytes.

8974 6.10.5 AES-CBC

8975 AES-CBC, denoted **CKM_AES_CBC**, is a mechanism for single- and multiple-part encryption and
8976 decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard and
8977 cipher-block chaining mode.

8978 It has a parameter, a 16-byte initialization vector.

8979 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
8980 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
8981 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
8982 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
8983 length as the padded input data. It does not wrap the key type, key length, or any other information about
8984 the key; the application must convey these separately.

8985 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
8986 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
8987 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
8988 attribute of the new key; other attributes required by the key type must be specified in the template.

8989 Constraints on key types and the length of data are summarized in the following table:

8990 Table 103, 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	

8991 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8992 specify the supported range of AES key sizes, in bytes.

8993 6.10.6 AES-CBC with PKCS padding

8994 AES-CBC with PKCS padding, denoted **CKM_AES_CBC_PAD**, is a mechanism for single- and multiple-
8995 part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced

8996 Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in PKCS
8997 #7.

8998 It has a parameter, a 16-byte initialization vector.

8999 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
9000 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified
9001 for the **CKA_VALUE_LEN** attribute.

9002 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
9003 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for
9004 details). The entries in the table below for data length constraints when wrapping and unwrapping keys
9005 do not apply to wrapping and unwrapping private keys.

9006 Constraints on key types and the length of data are summarized in the following table:

9007 *Table 104, 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

9008 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9009 specify the supported range of AES key sizes, in bytes.

9010 6.10.7 AES-OFB

9011 AES-OFB, denoted CKM_AES_OFB. It is a mechanism for single and multiple-part encryption and
9012 decryption with AES. AES-OFB mode is described in [NIST sp800-38a].

9013 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
9014 the block size.

9015

9016 Constraints on key types and the length of data are summarized in the following table:

9017

9018 *Table 105, 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

9019 For this mechanism the CK_MECHANISM_INFO structure is as specified for CBC mode.

9020 6.10.8 AES-CFB

9021 Cipher AES has a cipher feedback mode, AES-CFB, denoted CKM_AES_CFB8, CKM_AES_CFB64, and
9022 CKM_AES_CFB128. It is a mechanism for single and multiple-part encryption and decryption with AES.
9023 AES-OFB mode is described [NIST sp800-38a].

9024 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
9025 the block size.

9026

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

Table 106, 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

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

6.10.9 General-length AES-MAC

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

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

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

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

Table 107, 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

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

6.10.10 AES-MAC

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

It does not have a parameter.

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

Table 108, 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)

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

6.10.11 AES-XCBC-MAC

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

It does not have a parameter.

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

Table 109, 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

9057 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9058 specify the supported range of AES key sizes, in bytes.

9059 6.10.12 AES-XCBC-MAC-96

9060 AES-XCBC-MAC-96, denoted **CKM_AES_XCBC_MAC_96**, is a mechanism for single and multiple part
9061 signatures and verification; based on NIST's Advanced Encryption Standard and [RFC 3566].

9062 It does not have a parameter.

9063 Constraints on key types and the length of data are summarized in the following table:

9064 Table 110, 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

9065 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9066 specify the supported range of AES key sizes, in bytes.

9067 6.11 AES with Counter

9068 Table 111, 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	✓					✓	

9069 6.11.1 Definitions

9070 Mechanisms:

9071 CKM_AES_CTR

9072 6.11.2 AES with Counter mechanism parameters

9073 ♦ CK_AES_CTR_PARAMS; CK_AES_CTR_PARAMS_PTR

9074 **CK_AES_CTR_PARAMS** is a structure that provides the parameters to the **CKM_AES_CTR** mechanism.
9075 It is defined as follows:

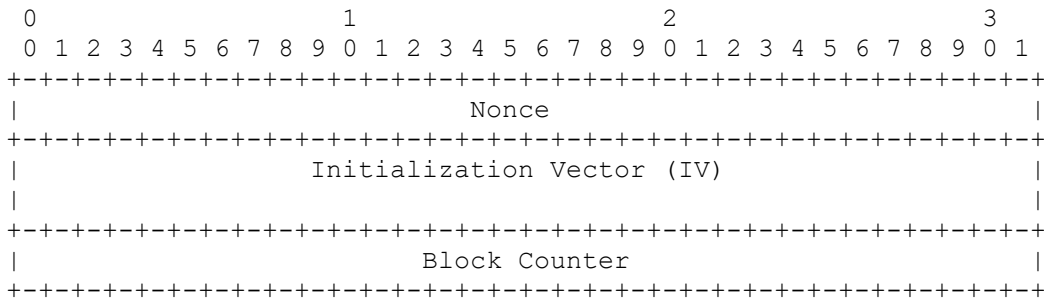
```
9076     typedef struct CK_AES_CTR_PARAMS {
9077         CK_ULONG    ulCounterBits;
9078         CK_BYTE     cb[16];
9079     } CK_AES_CTR_PARAMS;
```

9080

9081 *ulCounterBits* specifies the number of bits in the counter block (cb) that shall be incremented. This
9082 number shall be such that $0 < ulCounterBits \leq 128$. For any values outside this range the mechanism
9083 shall return **CKR_MECHANISM_PARAM_INVALID**.

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

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



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

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

6.11.3 AES with Counter Encryption / Decryption

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

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

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

6.12 AES CBC with Cipher Text Stealing CTS

Ref [NIST AES CTS]

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

Table 112, 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	✓					✓	

6.12.1 Definitions

Mechanisms:

CKM_AES_CTS

6.12.2 AES CTS mechanism parameters

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

Table 113, AES-CTS: Key And Data Length

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

6.13 Additional AES Mechanisms

Table 114, 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		✓					

6.13.1 Definitions

Mechanisms:

CKM_AES_GCM

CKM_AES_CCM

CKM_AES_GMAC

Generator Functions:

CKG_NO_GENERATE

CKG_GENERATE

CKG_GENERATE_COUNTER

CKG_GENERATE_RANDOM

CKG_GENERATE_COUNTER_XOR

6.13.2 AES-GCM Authenticated Encryption / Decryption

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

Encrypt:

- Set the IV length $u||vLen$ in the parameter block.
- Set the IV data $p||v$ in the parameter block.

- 9149 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
9150 *ulAADLen* is 0.
- 9151 • Set the tag length *ulTagBits* in the parameter block.
- 9152 • Call C_EncryptInit() for **CKM_AES_GCM** mechanism with parameters and key *K*.
- 9153 • Call C_Encrypt(), or C_EncryptUpdate()⁴ C_EncryptFinal(), for the plaintext obtaining ciphertext
9154 and authentication tag output.
- 9155 Decrypt:
- 9156 • Set the IV length *ullvLen* in the parameter block.
- 9157 • Set the IV data *pIv* in the parameter block.
- 9158 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
9159 *ulAADLen* is 0.
- 9160 • Set the tag length *ulTagBits* in the parameter block.
- 9161 • Call C_DecryptInit() for **CKM_AES_GCM** mechanism with parameters and key *K*.
- 9162 • Call C_Decrypt(), or C_DecryptUpdate()¹ C_DecryptFinal(), for the ciphertext, including the
9163 appended tag, obtaining plaintext output. Note: since **CKM_AES_GCM** is an AEAD cipher, no data
9164 should be returned until C_Decrypt() or C_DecryptFinal().
- 9165 MessageEncrypt:
- 9166 • Set the IV length *ullvLen* in the parameter block.
- 9167 • Set *pIv* to hold the IV data returned from C_EncryptMessage() and C_EncryptMessageBegin(). If
9168 *ullvFixedBits* is not zero, then the most significant bits of *pIv* contain the fixed IV. If *ivGenerator* is
9169 set to CKG_NO_GENERATE, *pIv* is an input parameter with the full IV.
- 9170 • Set the *ullvFixedBits* and *ivGenerator* fields in the parameter block.
- 9171 • Set the tag length *ulTagBits* in the parameter block.
- 9172 • Set *pTag* to hold the tag data returned from C_EncryptMessage() or the final
9173 C_EncryptMessageNext().
- 9174 • Call C_MessageEncryptInit() for **CKM_AES_GCM** mechanism key *K*.
- 9175 • Call C_EncryptMessage(), or C_EncryptMessageBegin() followed by C_EncryptMessageNext()⁵.
9176 The mechanism parameter is passed to all three of these functions.
- 9177 • Call C_MessageEncryptFinal() to close the message decryption.
- 9178 MessageDecrypt:
- 9179 • Set the IV length *ullvLen* in the parameter block.
- 9180 • Set the IV data *pIv* in the parameter block.
- 9181 • The *ullvFixedBits* and *ivGenerator* fields are ignored.
- 9182 • Set the tag length *ulTagBits* in the parameter block.
- 9183 • Set the tag data *pTag* in the parameter block before C_DecryptMessage() or the final
9184 C_DecryptMessageNext().

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

⁵ "*" indicates 0 or more calls may be made as required

9185 • Call C_MessageDecryptInit() for **CKM_AES_GCM** mechanism key *K*.

9186 • Call C_DecryptMessage(), or C_DecryptMessageBegin followed by C_DecryptMessageNext()^{*6}.

9187 The mechanism parameter is passed to all three of these functions.

9188 • Call C_MessageDecryptFinal() to close the message decryption.

9189 In *pIV* the least significant bit of the initialization vector is the rightmost bit. *ulIVLen* is the length of the

9190 initialization vector in bytes.

9191 On MessageEncrypt, the meaning of *ivGenerator* is as follows: CKG_NO_GENERATE means the IV is

9192 passed in on MessageEncrypt and no internal IV generation is done. CKG_GENERATE means that the

9193 non-fixed portion of the IV is generated by the module internally. The generation method is not defined.

9194 CKG_GENERATE_COUNTER means that the non-fixed portion of the IV is generated by the module

9195 internally by use of an incrementing counter, the initial IV counter is zero.

9196 CKG_GENERATE_COUNTER_XOR means that the non-fixed portion of the IV is xored with a counter.

9197 The value of the non-fixed portion passed must not vary from call to call. Like

9198 CKG_GENERATE_COUNTER, the counter starts at zero.

9199 CKG_GENERATE_RANDOM means that the non-fixed portion of the IV is generated by the module

9200 internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in *pIV*.

9201 Modules must implement CKG_GENERATE. Modules may also reject *ulIVFixedBits* values which are too

9202 large. Zero is always an acceptable value for *ulIVFixedBits*.

9203 In Encrypt and Decrypt the tag is appended to the cipher text and the least significant bit of the tag is the

9204 rightmost bit and the tag bits are the rightmost *ulTagBits* bits. In MessageEncrypt the tag is returned in

9205 the *pTag* field of CK_GCM_MESSAGE_PARAMS. In MessageDecrypt the tag is provided by the *pTag*

9206 field of CK_GCM_MESSAGE_PARAMS.

9207 The key type for *K* must be compatible with **CKM_AES_ECB** and the

9208 C_EncryptInit()/C_DecryptInit()/C_MessageEncryptInit()/C_MessageDecryptInit() calls shall behave, with

9209 respect to *K*, as if they were called directly with **CKM_AES_ECB**, *K* and NULL parameters.

9210 6.13.3 AES-CCM authenticated Encryption / Decryption

9211 For IPsec (RFC 4309) and also for use in ZFS encryption. Generic CCM mode is described in [RFC

9212 3610].

9213 To set up for AES-CCM use the following process, where *K* (key), nonce and additional authenticated

9214 data are as described in [RFC 3610]. AES-CCM uses CK_CCM_PARAMS for Encrypt and Decrypt, and

9215 CK_CCM_MESSAGE_PARAMS for MessageEncrypt and MessageDecrypt.

9216 Encrypt:

9217 • Set the message/data length *ulDataLen* in the parameter block.

9218 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.

9219 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if

9220 *ulAADLen* is 0.

9221 • Set the MAC length *ulMACLen* in the parameter block.

9222 • Call C_EncryptInit() for **CKM_AES_CCM** mechanism with parameters and key *K*.

9223 • Call C_Encrypt(), C_EncryptUpdate(), or C_EncryptFinal(), for the plaintext obtaining the final

9224 ciphertext output and the MAC. The total length of data processed must be *ulDataLen*. The output

9225 length will be *ulDataLen* + *ulMACLen*.

9226 Decrypt:

⁶ "*" indicates 0 or more calls may be made as required

- 9227 • Set the message/data length *ulDataLen* in the parameter block. This length must not include the
- 9228 length of the MAC that is appended to the cipher text.
- 9229 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- 9230 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
- 9231 *ulAADLen* is 0.
- 9232 • Set the MAC length *ulMACLen* in the parameter block.
- 9233 • Call C_DecryptInit() for **CKM_AES_CCM** mechanism with parameters and key *K*.
- 9234 • Call C_Decrypt(), C_DecryptUpdate(), or C_DecryptFinal(), for the ciphertext, including the
- 9235 appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen*
- 9236 + *ulMACLen*. Note: since **CKM_AES_CCM** is an AEAD cipher, no data should be returned until
- 9237 C_Decrypt() or C_DecryptFinal().
- 9238 MessageEncrypt:
- 9239 • Set the message/data length *ulDataLen* in the parameter block.
- 9240 • Set the nonce length *ulNonceLen*.
- 9241 • Set *pNonce* to hold the nonce data returned from C_EncryptMessage() and
- 9242 C_EncryptMessageBegin(). If *ulNonceFixedBits* is not zero, then the most significant bits of *pNonce*
- 9243 contain the fixed nonce. If *nonceGenerator* is set to CKG_NO_GENERATE, *pNonce* is an input
- 9244 parameter with the full nonce.
- 9245 • Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.
- 9246 • Set the MAC length *ulMACLen* in the parameter block.
- 9247 • Set *pMAC* to hold the MAC data returned from C_EncryptMessage() or the final
- 9248 C_EncryptMessageNext().
- 9249 • Call C_MessageEncryptInit() for **CKM_AES_CCM** mechanism key *K*.
- 9250 • Call C_EncryptMessage(), or C_EncryptMessageBegin() followed by C_EncryptMessageNext()^{*7}.
- 9251 The mechanism parameter is passed to all three functions.
- 9252 • Call C_MessageEncryptFinal() to close the message encryption.
- 9253 • The MAC is returned in *pMac* of the CK_CCM_MESSAGE_PARAMS structure.
- 9254 MessageDecrypt:
- 9255 • Set the message/data length *ulDataLen* in the parameter block.
- 9256 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block
- 9257 • The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.
- 9258 • Set the MAC length *ulMACLen* in the parameter block.
- 9259 • Set the MAC data *pMAC* in the parameter block before C_DecryptMessage() or the final
- 9260 C_DecryptMessageNext().
- 9261 • Call C_MessageDecryptInit() for **CKM_AES_CCM** mechanism key *K*.
- 9262 • Call C_DecryptMessage(), or C_DecryptMessageBegin() followed by C_DecryptMessageNext()^{*8}.
- 9263 The mechanism parameter is passed to all three functions.

⁷ "*" indicates 0 or more calls may be made as required

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

9264 • Call `C_MessageDecryptFinal()` to close the message decryption.

9265 In *pNonce* the least significant bit of the nonce is the rightmost bit. *ulNonceLen* is the length of the nonce

9266 in bytes.

9267 On `MessageEncrypt`, the meaning of *nonceGenerator* is as follows: `CKG_NO_GENERATE` means the

9268 nonce is passed in on `MessageEncrypt` and no internal MAC generation is done. `CKG_GENERATE`

9269 means that the non-fixed portion of the nonce is generated by the module internally. The generation

9270 method is not defined.

9271 `CKG_GENERATE_COUNTER` means that the non-fixed portion of the nonce is generated by the module

9272 internally by use of an incrementing counter, the initial IV counter is zero.

9273 `CKG_GENERATE_COUNTER_XOR` means that the non-fixed portion of the IV is xored with a counter.

9274 The value of the non-fixed portion passed must not vary from call to call. Like

9275 `CKG_GENERATE_COUNTER`, the counter starts at zero.

9276 `CKG_GENERATE_RANDOM` means that the non-fixed portion of the nonce is generated by the module

9277 internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.

9278 Modules must implement `CKG_GENERATE`. Modules may also reject *ulNonceFixedBits* values which are

9279 too large. Zero is always an acceptable value for *ulNonceFixedBits*.

9280 In `Encrypt` and `Decrypt` the MAC is appended to the cipher text and the least significant byte of the MAC

9281 is the rightmost byte and the MAC bytes are the rightmost *ulMACLen* bytes. In `MessageEncrypt` the MAC

9282 is returned in the *pMAC* field of `CK_CCM_MESSAGE_PARAMS`. In `MessageDecrypt` the MAC is

9283 provided by the *pMAC* field of `CK_CCM_MESSAGE_PARAMS`.

9284 The key type for K must be compatible with **CKM_AES_ECB** and the

9285 `C_EncryptInit()/C_DecryptInit()/C_MessageEncryptInit()/C_MessageDecryptInit()` calls shall behave, with

9286 respect to K, as if they were called directly with **CKM_AES_ECB**, K and NULL parameters.

9287 6.13.4 AES-GMAC

9288 AES-GMAC, denoted **CKM_AES_GMAC**, is a mechanism for single and multiple-part signatures and

9289 verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of

9290 GCM that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism

9291 parameters. When GMAC is used with `C_Sign` or `C_Verify`, *pData* points to the AAD. GMAC does not

9292 use plaintext or ciphertext.

9293 The signature produced by GMAC, also referred to as a Tag, the tag's length is determined by the

9294 `CK_GCM_PARAMS` field *ulTagBits*.

9295 The IV length is determined by the `CK_GCM_PARAMS` field *ullvLen*.

9296 Constraints on key types and the length of data are summarized in the following table:

9297 *Table 115, AES-GMAC: Key And Data Length*

Function	Key type	Data length	Signature length
<code>C_Sign</code>	<code>CKK_AES</code>	< 2 ⁶⁴	Depends on param's <i>ulTagBits</i>
<code>C_Verify</code>	<code>CKK_AES</code>	< 2 ⁶⁴	Depends on param's <i>ulTagBits</i>

9298 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure

9299 specify the supported range of AES key sizes, in bytes.

9300 6.13.5 AES GCM and CCM Mechanism parameters

9301 ♦ **CK_GENERATOR_FUNCTION**

9302 Functions to generate unique IVs and nonces.

9303 typedef CK_ULONG CK_GENERATOR_FUNCTION;

9304 ◆ **CK_GCM_PARAMS; CK_GCM_PARAMS_PTR**

9305 CK_GCM_PARAMS is a structure that provides the parameters to the CKM_AES_GCM mechanism
9306 when used for Encrypt or Decrypt. It is defined as follows:

```
9307       typedef struct CK_GCM_PARAMS {  
9308           CK_BYTE_PTR     pIv;  
9309           CK_ULONG        ulIvLen;  
9310           CK_ULONG        ulIvBits;  
9311           CK_BYTE_PTR     pAAD;  
9312           CK_ULONG        ulAADLen;  
9313           CK_ULONG        ulTagBits;  
9314       } CK_GCM_PARAMS;
```

9315

9316 The fields of the structure have the following meanings:

9317	plv	pointer to initialization vector
9318	ulIvLen	length of initialization vector in bytes. The length of the initialization
9319		vector can be any number between 1 and $(2^{32}) - 1$. 96-bit (12
9320		byte) IV values can be processed more efficiently, so that length is
9321		recommended for situations in which efficiency is critical.
9322	ulIvBits	length of initialization vector in bits. Do not use ulIvBits to specify the
9323		length of the initialization vector, but ulIvLen instead.
9324	pAAD	pointer to additional authentication data. This data is authenticated
9325		but not encrypted.
9326	ulAADLen	length of pAAD in bytes. The length of the AAD can be any number
9327		between 0 and $(2^{32}) - 1$.
9328	ulTagBits	length of authentication tag (output following cipher text) in bits. Can
9329		be any value between 0 and 128.

9330 **CK_GCM_PARAMS_PTR** is a pointer to a **CK_GCM_PARAMS**.

9331 ◆ **CK_GCM_MESSAGE_PARAMS; CK_GCM_MESSAGE_PARAMS_PTR**

9332 CK_GCM_MESSAGE_PARAMS is a structure that provides the parameters to the CKM_AES_GCM
9333 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```
9334       typedef struct CK_GCM_MESSAGE_PARAMS {  
9335           CK_BYTE_PTR     pIv;  
9336           CK_ULONG        ulIvLen;  
9337           CK_ULONG        ulIvFixedBits;  
9338           CK_GENERATOR_FUNCTION   ivGenerator;  
9339           CK_BYTE_PTR     pTag;  
9340           CK_ULONG        ulTagBits;  
9341       } CK_GCM_MESSAGE_PARAMS;
```

9342

9343 The fields of the structure have the following meanings:

9344	plv	pointer to initialization vector
------	-----	----------------------------------

9345	ulIvLen	length of initialization vector in bytes. The length of the initialization
9346		vector can be any number between 1 and $(2^{32}) - 1$. 96-bit (12 byte)
9347		IV values can be processed more efficiently, so that length is
9348		recommended for situations in which efficiency is critical.
9349	ulIvFixedBits	number of bits of the original IV to preserve when generating an
9350		new IV. These bits are counted from the Most significant bits (to the
9351		right).
9352	ivGenerator	Function used to generate a new IV. Each IV must be unique for a
9353		given session.
9354	pTag	location of the authentication tag which is returned on
9355		MessageEncrypt, and provided on MessageDecrypt.
9356	ulTagBits	length of authentication tag in bits. Can be any value between 0 and
9357		128.

9358 **CK_GCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_GCM_MESSAGE_PARAMS**.

9359

9360 ♦ **CK_CCM_PARAMS; CK_CCM_PARAMS_PTR**

9361 **CK_CCM_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism
9362 when used for Encrypt or Decrypt. It is defined as follows:

```

9363     typedef struct CK_CCM_PARAMS {
9364         CK_ULONG      ulDataLen; /*plaintext or ciphertext*/
9365         CK_BYTE_PTR    pNonce;
9366         CK_ULONG      ulNonceLen;
9367         CK_BYTE_PTR    pAAD;
9368         CK_ULONG      ulAADLen;
9369         CK_ULONG      ulMACLen;
9370     } CK_CCM_PARAMS;

```

9371 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
9372 length ($2 \leq L \leq 8$):

9373	ulDataLen	length of the data where $0 \leq \text{ulDataLen} < 2^{(8L)}$.
9374	pNonce	the nonce.
9375	ulNonceLen	length of pNonce in bytes where $7 \leq \text{ulNonceLen} \leq 13$.
9376	pAAD	Additional authentication data. This data is authenticated but not
9377		encrypted.
9378	ulAADLen	length of pAAD in bytes where $0 \leq \text{ulAADLen} \leq (2^{32}) - 1$.
9379	ulMACLen	length of the MAC (output following cipher text) in bytes. Valid
9380		values are 4, 6, 8, 10, 12, 14, and 16.

9381 **CK_CCM_PARAMS_PTR** is a pointer to a **CK_CCM_PARAMS**.

9382 ♦ **CK_CCM_MESSAGE_PARAMS; CK_CCM_MESSAGE_PARAMS_PTR**

9383 **CK_CCM_MESSAGE_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM**
9384 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```

9385     typedef struct CK_CCM_MESSAGE_PARAMS {
9386         CK_ULONG      ulDataLen; /*plaintext or ciphertext*/
9387         CK_BYTE_PTR    pNonce;

```

```

9388     CK_ULONG      ulNonceLen;
9389     CK_ULONG      ulNonceFixedBits;
9390     CK_GENERATOR_FUNCTION  nonceGenerator;
9391     CK_BYTE_PTR    pMAC;
9392     CK_ULONG      ulMACLen;
9393 } CK_CCM_MESSAGE_PARAMS;

```

9394

9395 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
 9396 length ($2 \leq L \leq 8$):

9397	ulDataLen	length of the data where $0 \leq \text{ulDataLen} < 2^{(8L)}$.
9398	pNonce	the nonce.
9399	ulNonceLen	length of pNonce in bytes where $7 \leq \text{ulNonceLen} \leq 13$.
9400	ulNonceFixedBits	number of bits of the original nonce to preserve when generating a
9401		new nonce. These bits are counted from the Most significant bits (to
9402		the right).
9403	nonceGenerator	Function used to generate a new nonce. Each nonce must be
9404		unique for a given session.
9405	pMAC	location of the CCM MAC returned on MessageEncrypt, provided on
9406		MessageDecrypt
9407	ulMACLen	length of the MAC (output following cipher text) in bytes. Valid
9408		values are 4, 6, 8, 10, 12, 14, and 16.

9409 **CK_CCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_CCM_MESSAGE_PARAMS**.

9410

9411 6.14 AES CMAC

9412 *Table 116, Mechanisms vs. Functions*

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

9413 ¹ SR = SignRecover, VR = VerifyRecover

9414 6.14.1 Definitions

9415 Mechanisms:

```

9416     CKM_AES_CMACHENERAL
9417     CKM_AES_CMACH

```

9418 6.14.2 Mechanism parameters

9419 CKM_AES_CMACHENERAL uses the existing **CK_MAC_GNERAL_PARAMS** structure.

9420 CKM_AES_CMACH does not use a mechanism parameter.

6.14.3 General-length AES-CMAC

General-length AES-CMAC, denoted **CKM_AES_CMACH**, is a mechanism for single- and multiple-part signatures and verification, based on [NIST SP800-38B] and [RFC 4493].

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

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

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

Table 117, 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

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

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

6.14.4 AES-CMAC

AES-CMAC, denoted **CKM_AES_CMACH**, is a special case of the general-length AES-CMAC mechanism. AES-MAC always produces and verifies MACs that are a full block size in length, the default output length specified by [RFC 4493].

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

Table 118, 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)

References [NIST SP800-38B] and [RFC 4493] recommend that the output MAC is not truncated to less than 64 bits. The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

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

6.15 AES XTS

Table 119, Mechanisms vs. Functions

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

6.15.1 Definitions

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

Mechanisms:

CKM_AES_XTS

CKM_AES_XTS_KEY_GEN

6.15.2 AES-XTS secret key objects

Table 120, 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

Refer to Table 11 for footnotes

6.15.3 AES-XTS key generation

The double-length AES-XTS key generation mechanism, denoted **CKM_AES_XTS_KEY_GEN**, is a key generation mechanism for double-length AES-XTS keys.

The mechanism generates AES-XTS keys with a particular length in bytes as specified in the CKA_VALUE_LEN attributes of the template for the key.

This mechanism contributes the CKA_CLASS, CKA_KEY_TYPE, and CKA_VALUE attributes to the new key. Other attributes supported by the double-length AES-XTS 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.

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

6.15.4 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 121, 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

6.16 AES Key Wrap

Table 122, AES Key Wrap Mechanisms vs. Functions

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	✓					✓	
CKM_AES_KEY_WRAP_PKCS7	✓					✓	
¹ SR = SignRecover, VR = VerifyRecover							

6.16.1 Definitions

Mechanisms:

- CKM_AES_KEY_WRAP
- CKM_AES_KEY_WRAP_PAD
- CKM_AES_KEY_WRAP_KWP
- CKM_AES_KEY_WRAP_PKCS7

6.16.2 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_PKCS7, 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 and CKM_AES_KEY_WRAP_PKCS7, resp. 4 for CKM_AES_KEY_WRAP_KWP, and the pointer non-NULL.

6.16.3 AES Key Wrap

The mechanisms support only single-part operations, i.e. single part wrapping and unwrapping, and single-part encryption and decryption.

◆ CKM_AES_KEY_WRAP

The CKM_AES_KEY_WRAP mechanism can wrap a key of any length. A secret key whose length is not a multiple of the AES Key Wrap semiblock size (8 bytes) will be zero padded to fit. Semiblock size is defined in Section 5.2 of [AES KEYWRAP]. A private key will be encoded as defined in section 6.7; the encoded private key will be zero padded to fit if necessary.

The CKM_AES_KEY_WRAP mechanism can only encrypt a block of data whose size is an exact multiple of the AES Key Wrap algorithm semiblock size.

For unwrapping, the mechanism decrypts the wrapped key. In case of a secret key, it 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 length specified in the template must not be less than n-7 bytes, where n is the length of the wrapped key. In case of a private key, the mechanism parses the encoding as defined in section 6.7 and ignores trailing zero bytes.

9510 ◆ **CKM_AES_KEY_WRAP_PAD**

9511 The CKM_AES_KEY_WRAP_PAD mechanism is deprecated. CKM_AES_KEY_WRAP_KWP resp.
9512 CKM_AES_KEY_WRAP_PKCS7 shall be used instead.

9513 ◆ **CKM_AES_KEY_WRAP_KWP**

9514 The CKM_AES_KEY_WRAP_KWP mechanism can wrap a key or encrypt block of data of any length.
9515 The input is zero-padded and wrapped / encrypted as defined in Section 6.3 of [AES KEYWRAP], which
9516 produces same results as RFC 5649.

9517 ◆ **CKM_AES_KEY_WRAP_PKCS7**

9518 The CKM_AES_KEY_WRAP_PKCS7 mechanism can wrap a key or encrypt a block of data of any
9519 length. It does the padding detailed in PKCS #7 of inputs (keys or data blocks) up to a semiblock size to
9520 make it an exact multiple of AES Key Wrap algorithm semiblock size (8bytes), always producing
9521 wrapped output that is larger than the input key/data to be wrapped. This padding is done by the token
9522 before being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of
9523 data as defined in Section 6.2 of [AES KEYWRAP].

9524 **6.17 Key derivation by data encryption – DES & AES**

9525 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
9526 They are for use with the C_DeriveKey function.

9527 Table 123, 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							✓

9528 **6.17.1 Definitions**

9529 Mechanisms:

9530 CKM_DES_ECB_ENCRYPT_DATA
9531 CKM_DES_CBC_ENCRYPT_DATA
9532 CKM_DES3_ECB_ENCRYPT_DATA
9533 CKM_DES3_CBC_ENCRYPT_DATA
9534 CKM_AES_ECB_ENCRYPT_DATA
9535 CKM_AES_CBC_ENCRYPT_DATA

9536
9537 typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
9538 CK_BYTE iv[8];


```

9539     CK_BYTE_PTR    pData;
9540     CK_ULONG        length;
9541 }    CK_DES_CBC_ENCRYPT_DATA_PARAMS;
9542
9543 typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
9544        CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;
9545
9546 typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
9547     CK_BYTE          iv[16];
9548     CK_BYTE_PTR      pData;
9549     CK_ULONG          length;
9550 }    CK_AES_CBC_ENCRYPT_DATA_PARAMS;
9551
9552 typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
9553        CK_AES_CBC_ENCRYPT_DATA_PARAMS_PTR;

```

6.17.2 Mechanism Parameters

Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 6.43.2

Table 124, 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.

6.17.3 Mechanism Description

The mechanisms will function by performing the encryption over the data provided using the base key. The resulting cipher text shall be used to create the key value of the resulting key. If not all the cipher text is used then the part discarded will be from the trailing end (least significant bytes) of the cipher text data. The derived key shall be defined by the attribute template supplied but constrained by the length of cipher text available for the key value and other normal PKCS11 derivation constraints.

Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1 Key Derivation mechanism in section 6.20.5.

If the data is too short to make the requested key then the mechanism returns CKR_DATA_LEN_RANGE.

6.18 Double and Triple-length DES

Table 125, 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	✓					✓	
CKM_DES3_CBC_PAD	✓					✓	
CKM_DES3_MAC_GENERAL		✓					
CKM_DES3_MAC		✓					

6.18.1 Definitions

This section defines the key type “CKK_DES2” and “CKK_DES3” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_DES2_KEY_GEN
CKM_DES3_KEY_GEN
CKM_DES3_ECB
CKM_DES3_CBC
CKM_DES3_MAC
CKM_DES3_MAC_GENERAL
CKM_DES3_CBC_PAD

6.18.2 DES2 secret key objects

DES2 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES2**) hold double-length DES keys. The following table defines the DES2 secret key object attributes, in addition to the common attributes defined for this object class:

Table 126, DES2 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 16 bytes long)

Refer to Table 11 for footnotes

DES2 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (i.e., each of the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or unwrap a DES2 key with incorrect parity will return an error.

The following is a sample template for creating a double-length DES secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_DES2;
CK_UTF8CHAR label[] = "A DES2 secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
```

9598 {CKA_TOKEN, &true, sizeof(true)},
9599 {CKA_LABEL, label, sizeof(label)-1},
9600 {CKA_ENCRYPT, &true, sizeof(true)},
9601 {CKA_VALUE, value, sizeof(value)}
9602 };
9603

9604 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
9605 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
9606 the key type of the secret key object.

9607 **6.18.3 DES3 secret key objects**

9608 DES3 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES3**) hold triple-length DES
9609 keys. The following table defines the DES3 secret key object attributes, in addition to the common
9610 attributes defined for this object class:

9611 *Table 127, DES3 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 24 bytes long)

9612 Refer to Table 11 for footnotes

9613 DES3 keys must always have their parity bits properly set as described in FIPS PUB 46-3 (*i.e.*, each of
9614 the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or
9615 unwrap a DES3 key with incorrect parity will return an error.

9616 The following is a sample template for creating a triple-length DES secret key object:

9617 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
9618 CK_KEY_TYPE keyType = CKK_DES3;
9619 CK_UTF8CHAR label[] = "A DES3 secret key object";
9620 CK_BYTE value[24] = {...};
9621 CK_BBOOL true = CK_TRUE;
9622 CK_ATTRIBUTE template[] = {
9623 {CKA_CLASS, &class, sizeof(class)},
9624 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
9625 {CKA_TOKEN, &true, sizeof(true)},
9626 {CKA_LABEL, label, sizeof(label)-1},
9627 {CKA_ENCRYPT, &true, sizeof(true)},
9628 {CKA_VALUE, value, sizeof(value)}
9629 };

9630
9631 CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first three
9632 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
9633 the key type of the secret key object.

9634 **6.18.4 Double-length DES key generation**

9635 The double-length DES key generation mechanism, denoted **CKM_DES2_KEY_GEN**, is a key
9636 generation mechanism for double-length DES keys. The DES keys making up a double-length DES key
9637 both have their parity bits set properly, as specified in FIPS PUB 46-3.

9638 It does not have a parameter.

9639 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
9640 key. Other attributes supported by the double-length DES 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.

Double-length DES keys can be used with all the same mechanisms as triple-DES keys:
CKM_DES3_ECB, **CKM_DES3_CBC**, **CKM_DES3_CBC_PAD**, **CKM_DES3_MAC_GENERAL**, and
CKM_DES3_MAC. Triple-DES encryption with a double-length DES key is equivalent to encryption with a triple-length DES key with K1=K3 as specified in FIPS PUB 46-3.

When double-length DES keys are generated, it is token-dependent whether or not it is possible for either of the component DES keys to be “weak” or “semi-weak” keys.

6.18.5 Triple-length DES Order of Operations

Triple-length DES encryptions are carried out as specified in FIPS PUB 46-3: encrypt, decrypt, encrypt. Decryptions are carried out with the opposite three steps: decrypt, encrypt, decrypt. The mathematical representations of the encrypt and decrypt operations are as follows:

$$\text{DES3-E}(\{K1, K2, K3\}, P) = E(K3, D(K2, E(K1, P)))$$

$$\text{DES3-D}(\{K1, K2, K3\}, C) = D(K1, E(K2, D(K3, P)))$$

6.18.6 Triple-length DES in CBC Mode

Triple-length DES operations in CBC mode, with double or triple-length keys, are performed using outer CBC as defined in X9.52. X9.52 describes this mode as TCBC. The mathematical representations of the CBC encrypt and decrypt operations are as follows:

$$\text{DES3-CBC-E}(\{K1, K2, K3\}, P) = E(K3, D(K2, E(K1, P + I)))$$

$$\text{DES3-CBC-D}(\{K1, K2, K3\}, C) = D(K1, E(K2, D(K3, P))) + I$$

The value *I* is either an 8-byte initialization vector or the previous block of cipher text that is added to the current input block. The addition operation is used is addition modulo-2 (XOR).

6.18.7 DES and Triple length DES in OFB Mode

Table 128, DES and Triple Length DES in OFB Mode Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_DES_OFB64	✓						
CKM_DES_OFB8	✓						
CKM_DES_CFB64	✓						
CKM_DES_CFB8	✓						

Cipher DES has a output feedback mode, DES-OFB, denoted **CKM_DES_OFB8** and **CKM_DES_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.

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

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

Table 129, 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

9672 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

9673 6.18.8 DES and Triple length DES in CFB Mode

9674 Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**.
9675 It is a mechanism for single and multiple-part encryption and decryption with DES.

9676 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
9677 the block size.

9678 Constraints on key types and the length of data are summarized in the following table:

9679 Table 130, 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

9680 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

9681 6.19 Double and Triple-length DES CMAC

9682 Table 131, 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_CMACE_GENERAL		✓					
CKM_DES3_CMACE		✓					

9683 ¹ SR = SignRecover, VR = VerifyRecover.

9684 6.19.1 Definitions

9685 Mechanisms:

9686 CKM_DES3_CMACE_GENERAL

9687 CKM_DES3_CMACE

9688 6.19.2 Mechanism parameters

9689 CKM_DES3_CMACE_GENERAL uses the existing **CK_MAC_GENERAL_PARAMS** structure.

9690 CKM_DES3_CMACE does not use a mechanism parameter.

6.19.3 General-length DES3-MAC

General-length DES3-CMAC, denoted **CKM_DES3_CMAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

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

The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in the MACing process.

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

Table 132, 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

Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits (which means using the entire block for DES). The MAC length must be specified before the communication starts, and must not be changed during the lifetime of the key. It is the caller's responsibility to follow these rules.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used

6.19.4 DES3-CMAC

DES3-CMAC, denoted **CKM_DES3_CMAC**, is a special case of the general-length DES3-CMAC mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the DES3 block length is the minimum output length recommended by [NIST sp800-38b].

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

Table 133, 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)

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.20 SHA-1

Table 134, 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							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA_1_KEY_GEN					✓		

6.20.1 Definitions

This section defines the key type “CKK_SHA_1_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA_1
CKM_SHA_1_HMAC
CKM_SHA_1_HMAC_GENERAL
CKM_SHA1_KEY_DERIVATION
CKM_SHA_1_KEY_GEN

6.20.2 SHA-1 digest

The SHA-1 mechanism, denoted **CKM_SHA_1**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 160-bit message digest defined in FIPS PUB 180-2.

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 135, SHA-1: Data Length

Function	Input length	Digest length
C_Digest	any	20

6.20.3 General-length SHA-1-HMAC

The general-length SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC_GENERAL**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the SHA-1 hash function. The keys it uses are generic secret keys and CKK_SHA_1_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 1-20 (the output size of SHA-1 is 20 bytes). Signatures (MACs) produced by this mechanism will be taken from the start of the full 20-byte HMAC output.

Table 136, 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

6.20.4 SHA-1-HMAC

The SHA-1-HMAC mechanism, denoted **CKM_SHA_1_HMAC**, is a special case of the general-length SHA-1-HMAC mechanism in Section 6.20.3.

It has no parameter, and always produces an output of length 20.

6.20.5 SHA-1 key derivation

SHA-1 key derivation, denoted **CKM_SHA1_KEY_DERIVATION**, is a mechanism which provides the capability of deriving a secret key by digesting the value of another secret key with SHA-1.

The value of the base key is digested once, and the result is used to make the value of derived secret key.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be 20 bytes (the output size of SHA-1).
- 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 was 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, 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 than 20 bytes, such as DES3, an error is generated.

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.

6.20.6 SHA-1 HMAC key generation

The SHA-1-HMAC key generation mechanism, denoted **CKM_SHA_1_KEY_GEN**, is a key generation mechanism for NIST's SHA-1-HMAC.

It does not have a parameter.

The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA-1-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA_1_HMAC** key sizes, in bytes.

6.21 SHA-224

Table 137, SHA-224 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA224				✓			
CKM_SHA224_HMAC		✓					
CKM_SHA224_HMAC_GENERAL		✓					
CKM_SHA224_KEY_DERIVATION							✓
CKM_SHA224_KEY_GEN					✓		

6.21.1 Definitions

This section defines the key type “CKK_SHA224_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_SHA224
- CKM_SHA224_HMAC
- CKM_SHA224_HMAC_GENERAL
- CKM_SHA224_KEY_DERIVATION
- CKM_SHA224_KEY_GEN

6.21.2 SHA-224 digest

The SHA-224 mechanism, denoted **CKM_SHA224**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 224-bit message digest defined in FIPS PUB 180-4.

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 138, SHA-224: Data Length

Function	Input length	Digest length
C_Digest	any	28

6.21.3 General-length SHA-224-HMAC

The general-length SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism except that it uses the HMAC construction based

on the SHA-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK_SHA224_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-224 hash output.

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 1-28 (the output size of SHA-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC output.

Table 139, 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

6.21.4 SHA-224-HMAC

The SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC**, is a special case of the general-length SHA-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

6.21.5 SHA-224 key derivation

SHA-224 key derivation, denoted **CKM_SHA224_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5 except that it uses the SHA-224 hash function and the relevant length is 28 bytes.

6.21.6 SHA-224 HMAC key generation

The SHA-224-HMAC key generation mechanism, denoted **CKM_SHA224_KEY_GEN**, is a key generation mechanism for NIST's SHA224-HMAC.

It does not have a parameter.

The mechanism generates SHA224-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA224-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA224_HMAC** key sizes, in bytes.

6.22 SHA-256

Table 140, 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					✓		

6.22.1 Definitions

This section defines the key type “CKK_SHA256_HMAC” for type CK_KEY_TYPE as used in the CK_A_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA256

CKM_SHA256_HMAC

CKM_SHA256_HMAC_GENERAL

CKM_SHA256_KEY_DERIVATION

CKM_SHA256_KEY_GEN

6.22.2 SHA-256 digest

The SHA-256 mechanism, denoted **CKM_SHA256**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 256-bit message digest defined in FIPS PUB 180-2.

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 141, SHA-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

6.22.3 General-length SHA-256-HMAC

The general-length SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_SHA256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-256 hash output.

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 SHA-256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

Table 142, 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

6.22.4 SHA-256-HMAC

The SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC**, is a special case of the general-length SHA-256-HMAC mechanism in Section 6.22.3.

It has no parameter, and always produces an output of length 32.

6.22.5 SHA-256 key derivation

SHA-256 key derivation, denoted **CKM_SHA256_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-256 hash function and the relevant length is 32 bytes.

6.22.6 SHA-256 HMAC key generation

The SHA-256-HMAC key generation mechanism, denoted **CKM_SHA256_KEY_GEN**, is a key generation mechanism for NIST's SHA256-HMAC.

It does not have a parameter.

The mechanism generates SHA256-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA256-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA256_HMAC** key sizes, in bytes.

6.23 SHA-384

Table 143, 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					✓		

6.23.1 Definitions

This section defines the key type “CKK_SHA384_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

9891 CKM_SHA384
9892 CKM_SHA384_HMAC
9893 CKM_SHA384_HMAC_GENERAL
9894 CKM_SHA384_KEY_DERIVATION
9895 CKM_SHA384_KEY_GEN

9896 6.23.2 SHA-384 digest

9897 The SHA-384 mechanism, denoted **CKM_SHA384**, is a mechanism for message digesting, following the
9898 Secure Hash Algorithm with a 384-bit message digest defined in FIPS PUB 180-2.

9899 It does not have a parameter.

9900 Constraints on the length of input and output data are summarized in the following table. For single-part
9901 digesting, the data and the digest may begin at the same location in memory.

9902 Table 144, SHA-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

9903 6.23.3 General-length SHA-384-HMAC

9904 The general-length SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC_GENERAL**, is the
9905 same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC
9906 construction based on the SHA-384 hash function and length of the output should be in the range 1-48.

9907 The keys it uses are generic secret keys and CKK_SHA384_HMAC. FIPS-198 compliant tokens may
9908 require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.

9909 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
9910 output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198
9911 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).
9912 Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC
9913 output.

9914 Table 145, 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

9915

9916 6.23.4 SHA-384-HMAC

9917 The SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC**, is a special case of the general-length
9918 SHA-384-HMAC mechanism.

9919 It has no parameter, and always produces an output of length 48.

9920 6.23.5 SHA-384 key derivation

9921 SHA-384 key derivation, denoted **CKM_SHA384_KEY_DERIVATION**, is the same as the SHA-1 key
9922 derivation mechanism in Section 6.20.5, except that it uses the SHA-384 hash function and the relevant
9923 length is 48 bytes.

6.23.6 SHA-384 HMAC key generation

The SHA-384-HMAC key generation mechanism, denoted **CKM_SHA384_KEY_GEN**, is a key generation mechanism for NIST's SHA384-HMAC.

It does not have a parameter.

The mechanism generates SHA384-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA384-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA384_HMAC** key sizes, in bytes.

6.24 SHA-512

Table 146, 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					✓		

6.24.1 Definitions

This section defines the key type “CKK_SHA512_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA512

CKM_SHA512_HMAC

CKM_SHA512_HMAC_GENERAL

CKM_SHA512_KEY_DERIVATION

CKM_SHA512_KEY_GEN

6.24.2 SHA-512 digest

The SHA-512 mechanism, denoted **CKM_SHA512**, is a mechanism for message digesting, following the Secure Hash Algorithm with a 512-bit message digest defined in FIPS PUB 180-2.

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 147, SHA-512: Data Length

Function	Input length	Digest length
C_Digest	any	64

6.24.3 General-length SHA-512-HMAC

The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

The keys it uses are generic secret keys and **CKK_SHA512_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

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 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC output.

Table 148, 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

6.24.4 SHA-512-HMAC

The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length SHA-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

6.24.5 SHA-512 key derivation

SHA-512 key derivation, denoted **CKM_SHA512_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

6.24.6 SHA-512 HMAC key generation

The SHA-512-HMAC key generation mechanism, denoted **CKM_SHA512_KEY_GEN**, is a key generation mechanism for NIST's SHA512-HMAC.

It does not have a parameter.

The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA512-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA512_HMAC** key sizes, in bytes.

6.25 SHA-512/224

Table 149, SHA-512/224 Mechanisms vs. Functions

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen. Key/ Key Pair	Wrap & Unwra p	Deriv e
CKM_SHA512_224				✓			
CKM_SHA512_224_HMAC_GENERAL		✓					
CKM_SHA512_224_HMAC		✓					
CKM_SHA512_224_KEY_DERIVATION							✓
CKM_SHA512_224_KEY_GEN					✓		

6.25.1 Definitions

This section defines the key type “CKK_SHA512_224_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA512_224
CKM_SHA512_224_HMAC
CKM_SHA512_224_HMAC_GENERAL
CKM_SHA512_224_KEY_DERIVATION
CKM_SHA512_224_KEY_GEN

6.25.2 SHA-512/224 digest

The SHA-512/224 mechanism, denoted **CKM_SHA512_224**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 224 bits. **CKM_SHA512_224** is the same as **CKM_SHA512_T** with a parameter value of 224.

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 150, SHA-512/224: Data Length

Function	Input length	Digest length
C_Digest	any	28

6.25.3 General-length SHA-512/224-HMAC

The general-length SHA-512/224-HMAC mechanism, denoted **CKM_SHA512_224_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-512/224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and CKK_SHA512_224_HMAC. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA-512/224 hash output.

10014 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
10015 output. This length should be in the range 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198
10016 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).
10017 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC
10018 output.

10019 Table 151, 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

10020

10021 **6.25.4 SHA-512/224-HMAC**

10022 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_224_HMAC**, is a special case of the general-
10023 length SHA-512/224-HMAC mechanism.

10024 It has no parameter, and always produces an output of length 28.

10025 **6.25.5 SHA-512/224 key derivation**

10026 The SHA-512/224 key derivation, denoted **CKM_SHA512_224_KEY_DERIVATION**, is the same as the
10027 SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/224 hash function
10028 and the relevant length is 28 bytes.

10029 **6.25.6 SHA-512/224 HMAC key generation**

10030 The SHA-512/224-HMAC key generation mechanism, denoted **CKM_SHA512_224_KEY_GEN**, is a key
10031 generation mechanism for NIST's SHA512/224-HMAC.

10032 It does not have a parameter.

10033 The mechanism generates SHA512/224-HMAC keys with a particular length in bytes, as specified in the
10034 **CKA_VALUE_LEN** attribute of the template for the key.

10035 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10036 key. Other attributes supported by the SHA512/224-HMAC key type (specifically, the flags indicating
10037 which functions the key supports) may be specified in the template for the key, or else are assigned
10038 default initial values.

10039 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10040 specify the supported range of **CKM_SHA512_224_HMAC** key sizes, in bytes.

10041 **6.26 SHA-512/256**

10042 Table 152, SHA-512/256 Mechanisms vs. Functions

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen. Key/ Key Pair	Wrap & Unwra p	Deriv e
CKM_SHA512_256				✓			

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ₁	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_256_HMAC_GENERAL		✓					
CKM_SHA512_256_HMAC		✓					
CKM_SHA512_256_KEY_DERIVATION							✓
CKM_SHA512_256_KEY_GEN					✓		

6.26.1 Definitions

This section defines the key type “CKK_SHA512_256_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA512_256
CKM_SHA512_256_HMAC
CKM_SHA512_256_HMAC_GENERAL
CKM_SHA512_256_KEY_DERIVATION
CKM_SHA512_256_KEY_GEN

6.26.2 SHA-512/256 digest

The SHA-512/256 mechanism, denoted **CKM_SHA512_256**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to 256 bits. **CKM_SHA512_256** is the same as **CKM_SHA512_T** with a parameter value of 256.

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 153, SHA-512/256: Data Length

Function	Input length	Digest length
C_Digest	any	32

6.26.3 General-length SHA-512/256-HMAC

The general-length SHA-512/256-HMAC mechanism, denoted **CKM_SHA512_256_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-512/256 hash function and length of the output should be in the range 1-32. The keys it uses are generic secret keys and CKK_SHA512_256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA-512/256 hash output.

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 SHA-512/256 is 32 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length). Signatures (MACs) produced by this mechanism will be taken from the start of the full 32-byte HMAC output.

10073 Table 154, 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

10074

10075 6.26.4 SHA-512/256-HMAC

10076 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_256_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism.

10078 It has no parameter, and always produces an output of length 32.

10079 6.26.5 SHA-512/256 key derivation

10080 The SHA-512/256 key derivation, denoted **CKM_SHA512_256_KEY_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/256 hash function and the relevant length is 32 bytes.

10083 6.26.6 SHA-512/256 HMAC key generation

10084 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.

10086 It does not have a parameter.

10087 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.

10089 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.

10093 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.

10095 6.27 SHA-512/t

10096 Table 155, 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	Generate Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA512_T_KEY_GEN					✓		

6.27.1 Definitions

This section defines the key type “CKK_SHA512_T_HMAC” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SHA512_T
CKM_SHA512_T_HMAC
CKM_SHA512_T_HMAC_GENERAL
CKM_SHA512_T_KEY_DERIVATION
CKM_SHA512_T_KEY_GEN

6.27.2 SHA-512/t digest

The SHA-512/t mechanism, denoted **CKM_SHA512_T**, is a mechanism for message digesting, following the Secure Hash Algorithm defined in FIPS PUB 180-4, section 5.3.6. It is based on a 512-bit message digest with a distinct initial hash value and truncated to t bits.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of $0 \leq \lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$.

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 156, SHA-512/256: Data Length

Function	Input length	Digest length
C_Digest	any	$\lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$

6.27.3 General-length SHA-512/t-HMAC

The general-length SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.3, except that it uses the HMAC construction based on the SHA-512/t hash function and length of the output should be in the range $0 \leq \lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$.

6.27.4 SHA-512/t-HMAC

The SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC**, is a special case of the general-length SHA-512/t-HMAC mechanism.

It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in bytes of the desired output should be in the range of $0 \leq \lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$.

6.27.5 SHA-512/t key derivation

The SHA-512/t key derivation, denoted **CKM_SHA512_T_KEY_DERIVATION**, is the same as the SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/t hash function and the relevant length is $\lceil t/8 \rceil$ bytes, where $0 < t < 512$, and $t \neq 384$.

6.27.6 SHA-512/t HMAC key generation

The SHA-512/t-HMAC key generation mechanism, denoted **CKM_SHA512_T_KEY_GEN**, is a key generation mechanism for NIST's SHA512/t-HMAC.

It does not have a parameter.

The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA512/t-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA512_T_HMAC** key sizes, in bytes.

6.28 SHA3-224

Table 157, SHA3-224 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					✓		

6.28.1 Definitions

Mechanisms:

CKM_SHA3_224

CKM_SHA3_224_HMAC

CKM_SHA3_224_HMAC_GENERAL

CKM_SHA3_224_KEY_DERIVATION

CKM_SHA3_224_KEY_GEN

CKK_SHA3_224_HMAC

6.28.2 SHA3-224 digest

The SHA3-224 mechanism, denoted **CKM_SHA3_224**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 224-bit message digest defined in FIPS Pub 202.

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 158, SHA3-224: Data Length

Function	Input length	Digest length
C_Digest	any	28

6.28.3 General-length SHA3-224-HMAC

The general-length SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in section 6.20.4 except that it uses the HMAC construction based on the SHA3-224 hash function and length of the output should be in the range 1-28. The keys it uses are generic secret keys and **CKK_SHA3_224_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 14 bytes; that is, half the size of the SHA3-224 hash output.

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 1-28 (the output size of SHA3-224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 28-byte HMAC output.

Table 159, 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

6.28.4 SHA3-224-HMAC

The SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC**, is a special case of the general-length SHA3-224-HMAC mechanism.

It has no parameter, and always produces an output of length 28.

6.28.5 SHA3-224 key derivation

SHA-224 key derivation, denoted **CKM_SHA3_224_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5 except that it uses the SHA3-224 hash function and the relevant length is 28 bytes.

6.28.6 SHA3-224 HMAC key generation

The SHA3-224-HMAC key generation mechanism, denoted **CKM_SHA3_224_KEY_GEN**, is a key generation mechanism for NIST's SHA3-224-HMAC.

It does not have a parameter.

The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA3-224-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA3_224_HMAC** key sizes, in bytes.

6.29 SHA3-256

Table 160, 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					✓		

6.29.1 Definitions

- Mechanisms:
- CKM_SHA3_256
 - CKM_SHA3_256_HMAC
 - CKM_SHA3_256_HMAC_GENERAL
 - CKM_SHA3_256_KEY_DERIVATION
 - CKM_SHA3_256_KEY_GEN
- CKK_SHA3_256_HMAC

6.29.2 SHA3-256 digest

The SHA3-256 mechanism, denoted **CKM_SHA3_256**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 256-bit message digest defined in FIPS PUB 202.

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 161, SHA3-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

6.29.3 General-length SHA3-256-HMAC

The general-length SHA3-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC construction based on the SHA3-256 hash function and length of the output should be in the range 1-32.

The keys it uses are generic secret keys and CKK_SHA3_256_HMAC. FIPS-198 compliant tokens may require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.

10216 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
 10217 output. This length should be in the range 1-32 (the output size of SHA3-256 is 32 bytes). FIPS-198
 10218 compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
 10219 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC
 10220 output.

10221 *Table 162, 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

10222 6.29.4 SHA3-256-HMAC

10223 The SHA-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC**, is a special case of the general-
 10224 length SHA-256-HMAC mechanism.

10225 It has no parameter, and always produces an output of length 32.

10226 6.29.5 SHA3-256 key derivation

10227 SHA-256 key derivation, denoted **CKM_SHA3_256_KEY_DERIVATION**, is the same as the SHA-1 key
 10228 derivation mechanism in Section 6.20.5, except that it uses the SHA3-256 hash function and the relevant
 10229 length is 32 bytes.

10230 6.29.6 SHA3-256 HMAC key generation

10231 The SHA3-256-HMAC key generation mechanism, denoted **CKM_SHA3_256_KEY_GEN**, is a key
 10232 generation mechanism for NIST's SHA3-256-HMAC.

10233 It does not have a parameter.

10234 The mechanism generates SHA3-256-HMAC keys with a particular length in bytes, as specified in the
 10235 **CKA_VALUE_LEN** attribute of the template for the key.

10236 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 10237 key. Other attributes supported by the SHA3-256-HMAC key type (specifically, the flags indicating which
 10238 functions the key supports) may be specified in the template for the key, or else are assigned default
 10239 initial values.

10240 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10241 specify the supported range of **CKM_SHA3_256_HMAC** key sizes, in bytes.

10242

10243 6.30 SHA3-384

10244 *Table 163, SHA3-384 Mechanisms vs. Functions*

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_384				✓			
CKM_SHA3_384_HMAC_GENERAL		✓					
CKM_SHA3_384_HMAC		✓					
CKM_SHA3_384_KEY_DERIVATION							✓

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_SHA3_384_KEY_GEN				✓			

6.30.1 Definitions

CKM_SHA3_384
CKM_SHA3_384_HMAC
CKM_SHA3_384_HMAC_GENERAL
CKM_SHA3_384_KEY_DERIVATION
CKM_SHA3_384_KEY_GEN
CKK_SHA3_384_HMAC

6.30.2 SHA3-384 digest

The SHA3-384 mechanism, denoted **CKM_SHA3_384**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 384-bit message digest defined in FIPS PUB 202.

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 164, SHA3-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

6.30.3 General-length SHA3-384-HMAC

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 6.20.4, 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.

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.

Table 165, 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

6.30.4 SHA3-384-HMAC

The SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC**, is a special case of the general-length SHA3-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

6.30.5 SHA3-384 key derivation

SHA3-384 key derivation, denoted **CKM_SHA3_384_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-384 hash function and the relevant length is 48 bytes.

6.30.6 SHA3-384 HMAC key generation

The SHA3-384-HMAC key generation mechanism, denoted **CKM_SHA3_384_KEY_GEN**, is a key generation mechanism for NIST's SHA3-384-HMAC.

It does not have a parameter.

The mechanism generates SHA3-384-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA3-384-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA3_384_HMAC** key sizes, in bytes.

6.31 SHA3-512

Table 166, SHA-512 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR 1	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				✓			

6.31.1 Definitions

CKM_SHA3_512

CKM_SHA3_512_HMAC

CKM_SHA3_512_HMAC_GENERAL

CKM_SHA3_512_KEY_DERIVATION

CKM_SHA3_512_KEY_GEN

CKK_SHA3_512_HMAC

6.31.2 SHA3-512 digest

The SHA3-512 mechanism, denoted **CKM_SHA3_512**, is a mechanism for message digesting, following the Secure Hash 3 Algorithm with a 512-bit message digest defined in FIPS PUB 202.

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 167, SHA3-512: Data Length

Function	Input length	Digest length
C_Digest	any	64

6.31.3 General-length SHA3-512-HMAC

The general-length SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC_GENERAL**, is the same as the general-length SHA-1-HMAC mechanism in Section 6.20.4, except that it uses the HMAC construction based on the SHA3-512 hash function and length of the output should be in the range 1-64. The keys it uses are generic secret keys and **CKK_SHA3_512_HMAC**. FIPS-198 compliant tokens may require the key length to be at least 32 bytes; that is, half the size of the SHA3-512 hash output.

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 1-64 (the output size of SHA3-512 is 64 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

Table 168, 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

6.31.4 SHA3-512-HMAC

The SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC**, is a special case of the general-length SHA3-512-HMAC mechanism.

It has no parameter, and always produces an output of length 64.

6.31.5 SHA3-512 key derivation

SHA3-512 key derivation, denoted **CKM_SHA3_512_KEY_DERIVATION**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the SHA-512 hash function and the relevant length is 64 bytes.

6.31.6 SHA3-512 HMAC key generation

The SHA3-512-HMAC key generation mechanism, denoted **CKM_SHA3_512_KEY_GEN**, is a key generation mechanism for NIST's SHA3-512-HMAC.

It does not have a parameter.

The mechanism generates SHA3-512-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

10339 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 10340 key. Other attributes supported by the SHA3-512-HMAC key type (specifically, the flags indicating which
 10341 functions the key supports) may be specified in the template for the key, or else are assigned default
 10342 initial values.

10343 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10344 specify the supported range of **CKM_SHA3_512_HMAC** key sizes, in bytes.

10345 6.32 SHAKE

10346 *Table 169, 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							✓

10347 6.32.1 Definitions

10348 CKM_SHAKE_128_KEY_DERIVATION

10349 CKM_SHAKE_256_KEY_DERIVATION

10350 6.32.2 SHAKE Key Derivation

10351 SHAKE-128 and SHAKE-256 key derivation, denoted **CKM_SHAKE_128_KEY_DERIVATION** and
 10352 **CKM_SHAKE_256_KEY_DERIVATION**, implements the SHAKE expansion function defined in FIPS 202
 10353 on the input key.

- 10354 • If no length or key type is provided in the template a **CKR_TEMPLATE_INCOMPLETE** error is
 10355 generated.
- 10356 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
 10357 shall be a generic secret key of the specified length.
- 10358 • If no length was provided in the template, but a key type is, then that key type must have a well-
 10359 defined length. If it does, then the key produced by this mechanism shall be of the type specified in
 10360 the template. If it doesn't, an error shall be returned.
- 10361 • If both a key type and a length are provided in the template, the length must be compatible with that
 10362 key type. The key produced by this mechanism shall be of the specified type and length.

10363 If a DES, DES2, or CDMF key is derived with this mechanism, the parity bits of the key shall be set
 10364 properly.

10365 This mechanism has the following rules about key sensitivity and extractability:

- 10366 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
 10367 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
 10368 default value.
- 10369 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
 10370 shall as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
 10371 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 10372 **CKA_SENSITIVE** attribute.
- 10373 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then
 10374 the derived key shall, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 10375 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 10376 value from its **CKA_EXTRACTABLE** attribute.

10377 **6.33 BLAKE2B-160**

10378 *Table 170, 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					✓		

10379 **6.33.1 Definitions**

10380 Mechanisms:

- 10381 CKM_BLAKE2B_160
- 10382 CKM_BLAKE2B_160_HMAC
- 10383 CKM_BLAKE2B_160_HMAC_GENERAL
- 10384 CKM_BLAKE2B_160_KEY_DERIVE
- 10385 CKM_BLAKE2B_160_KEY_GEN
- 10386 CKK_BLAKE2B_160_HMAC

10387 **6.33.2 BLAKE2B-160 digest**

10388 The BLAKE2B-160 mechanism, denoted **CKM_BLAKE2B_160**, is a mechanism for message digesting,
10389 following the Blake2b Algorithm with a 160-bit message digest without a key as defined in [RFC 7693](#).

10390 It does not have a parameter.

10391 Constraints on the length of input and output data are summarized in the following table. For single-part
10392 digesting, the data and the digest may begin at the same location in memory.

10393 *Table 171, BLAKE2B-160: Data Length*

Function	Input length	Digest length
C_Digest	any	20

10394 **6.33.3 General-length BLAKE2B-160-HMAC**

10395 The general-length BLAKE2B-160-HMAC mechanism, denoted
10396 **CKM_BLAKE2B_160_HMAC_GENERAL**, is the keyed variant of BLAKE2b-160 and length of the output
10397 should be in the range 1-20. The keys it uses are generic secret keys and CKK_BLAKE2B_160_HMAC.

10398 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
10399 output. This length should be in the range 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures
10400 (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.

10401 *Table 172, 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

6.33.4 BLAKE2B-160-HMAC

The BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC**, is a special case of the general-length BLAKE2B-160-HMAC mechanism.

It has no parameter, and always produces an output of length 20.

6.33.5 BLAKE2B-160 key derivation

BLAKE2B-160 key derivation, denoted **CKM_BLAKE2B_160_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5 except that it uses the BLAKE2B-160 hash function and the relevant length is 20 bytes.

6.33.6 BLAKE2B-160 HMAC key generation

The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM_BLAKE2B_160_KEY_GEN**, is a key generation mechanism for BLAKE2B-160-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-160-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_BLAKE2B_160_HMAC** key sizes, in bytes.

6.34 BLAKE2B-256

Table 173, 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					✓		

6.34.1 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

6.34.2 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 174, BLAKE2B-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

6.34.3 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.

Table 175, General-length BLAKE2B-256-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_256_HMAC	Any	1-32, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_256_HMAC	Any	1-32, depending on parameters

6.34.4 BLAKE2B-256-HMAC

The BLAKE2B-256-HMAC mechanism, denoted **CKM_BLAKE2B_256_HMAC**, is a special case of the general-length BLAKE2B-256-HMAC mechanism in Section 6.34.3.

It has no parameter, and always produces an output of length 32.

6.34.5 BLAKE2B-256 key derivation

BLAKE2B-256 key derivation, denoted **CKM_BLAKE2B_256_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-256 hash function and the relevant length is 32 bytes.

6.34.6 BLAKE2B-256 HMAC key generation

The BLAKE2B-256-HMAC key generation mechanism, denoted **CKM_BLAKE2B_256_KEY_GEN**, is a key generation mechanism for BLAKE2B-256-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-256-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_BLAKE2B_256_HMAC** key sizes, in bytes.

6.35 BLAKE2B-384

Table 176, 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				✓			

6.35.1 Definitions

CKM_BLAKE2B_384

CKM_BLAKE2B_384_HMAC

CKM_BLAKE2B_384_HMAC_GENERAL

CKM_BLAKE2B_384_KEY_DERIVE

CKM_BLAKE2B_384_KEY_GEN

CKM_BLAKE2B_384_HMAC

6.35.2 BLAKE2B-384 digest

The BLAKE2B-384 mechanism, denoted **CKM_BLAKE2B_384**, is a mechanism for message digesting, following the Blake2b Algorithm with a 384-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 177, BLAKE2B-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

6.35.3 General-length BLAKE2B-384-HMAC

The general-length BLAKE2B-384-HMAC mechanism, denoted **CKM_BLAKE2B_384_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-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_BLAKE2B_384_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 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

Table 178, General-length BLAKE2B-384-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_384_HMAC	Any	1-48, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_384_HMAC	Any	1-48, depending on parameters

6.35.4 BLAKE2B-384-HMAC

The BLAKE2B-384-HMAC mechanism, denoted **CKM_BLAKE2B_384_HMAC**, is a special case of the general-length BLAKE2B-384-HMAC mechanism.

It has no parameter, and always produces an output of length 48.

6.35.5 BLAKE2B-384 key derivation

BLAKE2B-384 key derivation, denoted **CKM_BLAKE2B_384_KEY_DERIVE**, is the same as the SHA-1 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-384 hash function and the relevant length is 48 bytes.

6.35.6 BLAKE2B-384 HMAC key generation

The BLAKE2B-384-HMAC key generation mechanism, denoted **CKM_BLAKE2B_384_KEY_GEN**, is a key generation mechanism for NIST's BLAKE2B-384-HMAC.

It does not have a parameter.

The mechanism generates BLAKE2B-384-HMAC keys with a particular length in bytes, as specified in the **CKA_VALUE_LEN** attribute of the template for the key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the BLAKE2B-384-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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_BLAKE2B_384_HMAC** key sizes, in bytes.

6.36 BLAKE2B-512

Table 179, 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				✓			

6.36.1 Definitions

CKM_BLAKE2B_512
 CKM_BLAKE2B_512_HMAC
 CKM_BLAKE2B_512_HMAC_GENERAL
 CKM_BLAKE2B_512_KEY_DERIVE
 CKM_BLAKE2B_512_KEY_GEN
 CKK_BLAKE2B_512_HMAC

6.36.2 BLAKE2B-512 digest

The BLAKE2B-512 mechanism, denoted **CKM_BLAKE2B_512**, is a mechanism for message digesting, following the Blake2b Algorithm with a 512-bit message digest 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 180, BLAKE2B-512: Data Length

Function	Input length	Digest length
C_Digest	any	64

6.36.3 General-length BLAKE2B-512-HMAC

The general-length BLAKE2B-512-HMAC mechanism, denoted **CKM_BLAKE2B_512_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-512 hash function and length of the output should be in the range 1-64. The keys it uses are generic secret keys and CKK_BLAKE2B_512_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 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

Table 181, 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_HMAC	Any	1-64, depending on parameters

10540

10541 6.36.4 BLAKE2B-512-HMAC

10542 The BLAKE2B-512-HMAC mechanism, denoted **CKM_BLAKE2B_512_HMAC**, is a special case of the
10543 general-length BLAKE2B-512-HMAC mechanism.

10544 It has no parameter, and always produces an output of length 64.

10545 6.36.5 BLAKE2B-512 key derivation

10546 BLAKE2B-512 key derivation, denoted **CKM_BLAKE2B_512_KEY_DERIVE**, is the same as the SHA-1
10547 key derivation mechanism in Section 6.20.5, except that it uses the BLAKE2B-512 hash function and the
10548 relevant length is 64 bytes.

10549 6.36.6 BLAKE2B-512 HMAC key generation

10550 The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM_BLAKE2B_512_KEY_GEN**, is a
10551 key generation mechanism for NIST's BLAKE2B-512-HMAC.

10552 It does not have a parameter.

10553 The mechanism generates BLAKE2B-512-HMAC keys with a particular length in bytes, as specified in the
10554 **CKA_VALUE_LEN** attribute of the template for the key.

10555 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10556 key. Other attributes supported by the BLAKE2B-512-HMAC key type (specifically, the flags indicating
10557 which functions the key supports) may be specified in the template for the key, or else are assigned
10558 default initial values.

10559 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10560 specify the supported range of **CKM_BLAKE2B_512_HMAC** key sizes, in bytes.

10561

10562 6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

10563 The mechanisms in this section are for generating keys and IVs for performing password-based
10564 encryption. The method used to generate keys and IVs is specified in PKCS #5.

10565 *Table 182, 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					✓		

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen . Key/ Key Pair	Wrap & Unwra p	Deriv e
CKM_PKCS5_PBKD2					✓		

6.37.1 Definitions

Mechanisms:

CKM_PBE_SHA1_DES3_EDE_CBC

CKM_PBE_SHA1_DES2_EDE_CBC

CKM_PKCS5_PBKD2

CKM_PBA_SHA1_WITH_SHA1_HMAC

6.37.2 Password-based encryption/authentication mechanism parameters

◆ CK_PBE_PARAMS; CK_PBE_PARAMS_PTR

CK_PBE_PARAMS is a structure which provides all of the necessary information required by the CKM_PBE mechanisms (see PKCS #5 and PKCS #12 for information on the PBE generation mechanisms) and the CKM_PBA_SHA1_WITH_SHA1_HMAC mechanism. It is defined as follows:

```
typedef struct CK_PBE_PARAMS {
    CK_BYTE_PTR      pInitVector;
    CK_UTF8CHAR_PTR  pPassword;
    CK_ULONG         ulPasswordLen;
    CK_BYTE_PTR      pSalt;
    CK_ULONG         ulSaltLen;
    CK_ULONG         ulIteration;
} CK_PBE_PARAMS;
```

The fields of the structure have the following meanings:

pInitVector	pointer to the location that receives the 8-byte initialization vector (IV), if an IV is required;
pPassword	points to the password to be used in the PBE key generation;
ulPasswordLen	length in bytes of the password information;
pSalt	points to the salt to be used in the PBE key generation;
ulSaltLen	length in bytes of the salt information;
ulIteration	number of iterations required for the generation.

CK_PBE_PARAMS_PTR is a pointer to a **CK_PBE_PARAMS**.

6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters

◆ CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE; CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR

CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE is used to indicate the Pseudo-Random Function (PRF) used to generate key bits using PKCS #5 PBKDF2. It is defined as follows:

```
typedef CK_ULONG CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE;
```

The following PRFs are defined in PKCS #5 v2.1. The following table lists the defined functions.

Table 183, 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 NULL_PTR, 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.

CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR is a pointer to a **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE**.

10607 ♦ **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;**
10608 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR**

10609 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE** is used to indicate the source of the salt value when
10610 deriving a key using PKCS #5 PBKDF2. It is defined as follows:

10611 `typedef CK_ULONG CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;`
10612

10613 The following salt value sources are defined in PKCS #5 v2.1. The following table lists the defined
10614 sources along with the corresponding data type for the *pSaltSourceData* field in the
10615 **CK_PKCS5_PBKD2_PARAMS2** structure defined below.

10616 *Table 184, 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.

10617 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR** is a pointer to a
10618 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE**.

10619 ♦ **CK_PKCS5_PBKD2_PARAMS2; CK_PKCS5_PBKD2_PARAMS2_PTR**

10620 **CK_PKCS5_PBKD2_PARAMS2** is a structure that provides the parameters to the
10621 **CKM_PKCS5_PBKD2** mechanism. The structure is defined as follows:

10622 `typedef struct CK_PKCS5_PBKD2_PARAMS2 {`
10623 `CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE saltSource;`
10624 `CK_VOID_PTR pSaltSourceData;`
10625 `CK_ULONG ulSaltSourceDataLen;`
10626 `CK_ULONG iterations;`
10627 `CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE prf;`
10628 `CK_VOID_PTR pPrfData;`
10629 `CK_ULONG ulPrfDataLen;`
10630 `CK_UTF8CHAR_PTR pPassword;`
10631 `CK_ULONG ulPasswordLen;`
10632 `} CK_PKCS5_PBKD2_PARAMS2;`
10633

10634 The fields of the structure have the following meanings:

10635	<code>saltSource</code>	source of the salt value
10636	<code>pSaltSourceData</code>	data used as the input for the salt source
10637	<code>ulSaltSourceDataLen</code>	length of the salt source input
10638	<code>iterations</code>	number of iterations to perform when generating each block of
10639		random data
10640	<code>prf</code>	pseudo-random function used to generate the key
10641	<code>pPrfData</code>	data used as the input for PRF in addition to the salt value
10642	<code>ulPrfDataLen</code>	length of the input data for the PRF
10643	<code>pPassword</code>	points to the password to be used in the PBE key generation
10644	<code>ulPasswordLen</code>	length in bytes of the password information

10645 **CK_PKCS5_PBKD2_PARAMS2_PTR** is a pointer to a **CK_PKCS5_PBKD2_PARAMS2**.

6.37.4 PKCS #5 PBKD2 key generation

PKCS #5 PBKDF2 key generation, denoted **CKM_PKCS5_PBKD2**, is a mechanism used for generating a secret key from a password and a salt value. This functionality is defined in PKCS#5 as PBKDF2.

It has a parameter, a **CK_PKCS5_PBKD2_PARAMS2** structure. The parameter specifies the salt value source, pseudo-random function, and iteration count used to generate the new key.

Since this mechanism can be used to generate any type of secret key, new key templates must contain the **CKA_KEY_TYPE** and **CKA_VALUE_LEN** attributes. If the key type has a fixed length the **CKA_VALUE_LEN** attribute may be omitted.

6.38 PKCS #12 password-based encryption/authentication mechanisms

The mechanisms in this section are for generating keys and IVs for performing password-based encryption or authentication. The method used to generate keys and IVs is based on a method that was specified in PKCS #12.

We specify here a general method for producing various types of pseudo-random bits from a password, p ; a string of salt bits, s ; and an iteration count, c . The “type” of pseudo-random bits to be produced is identified by an identification byte, ID , the meaning of which will be discussed later.

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 variable and output of length u bits, and the message input to the compression function of H is v bits). For MD2 and MD5, $u=128$ and $v=512$; for SHA-1, $u=160$ and $v=512$.

We assume here that u and v are both multiples of 8, as are the lengths in bits of the password and salt strings and the number n of pseudo-random bits required. In addition, u and v are of course nonzero.

1. Construct a string, D (the “diversifier”), by concatenating $v/8$ copies of ID .
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 .
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 .
4. Set $I = S || P$ to be the concatenation of S and P .
5. Set $j = \lceil n/u \rceil$.
6. For $i=1, 2, \dots, j$, do the following:
 - 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.
 - b. Concatenate copies of A_i to create a string B of length v bits (the final copy of A_i may be truncated to create B).
 - 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 setting $I_j = (I_j + B + 1) \bmod 2^v$ for each j . To perform this addition, treat each v -bit block as a binary number represented most-significant bit first.
7. Concatenate A_1, A_2, \dots, A_j together to form a pseudo-random bit string, A .
8. Use the first n bits of A as the output of this entire process.

When the password-based encryption mechanisms presented in this section are used to generate a key and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To generate a key, the identifier byte ID is set to the value 1; to generate an IV, the identifier byte ID is set to the value 2.

When the password based authentication mechanism presented in this section is used to generate a key from a password, salt, and an iteration count, the above algorithm is used. The identifier byte ID is set to the value 3.

6.38.1 SHA-1-PBE for 3-key triple-DES-CBC

SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES3_EDE_CBC**, is a mechanism used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

6.38.2 SHA-1-PBE for 2-key triple-DES-CBC

SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES2_EDE_CBC**, is a mechanism used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-key triple-DES key with proper parity bits is obtained.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process and the location of the application-supplied buffer which will receive the 8-byte IV generated by the mechanism.

The key and IV produced by this mechanism will typically be used for performing password-based encryption.

6.38.3 SHA-1-PBA for SHA-1-HMAC

SHA-1-PBA for SHA-1-HMAC, denoted **CKM_PBA_SHA1_WITH_SHA1_HMAC**, is a mechanism used for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest algorithm and an iteration count. The method used to generate the key is described above.

It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the key generation process. The parameter also has a field to hold the location of an application-supplied buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since authentication with SHA-1-HMAC does not require an IV.

The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform password-based authentication (not *password-based encryption*). At the time of this writing, this is primarily done to ensure the integrity of a PKCS #12 PDU.

6.39 SSL

Table 185, SSL Mechanisms vs. Functions

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen . Key / Key Pair	Wrap & Unwra p	Deriv e
CKM_SSL3_PRE_MASTER_KEY_GEN					✓		
CKM_TLS_PRE_MASTER_KEY_GEN					✓		

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen · Key / Key Pair	Wrap & Unwra p	Deriv e
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		✓					

6.39.1 Definitions

Mechanisms:

CKM_SSL3_PRE_MASTER_KEY_GEN
CKM_TLS_PRE_MASTER_KEY_GEN
CKM_SSL3_MASTER_KEY_DERIVE
CKM_SSL3_KEY_AND_MAC_DERIVE
CKM_SSL3_MASTER_KEY_DERIVE_DH
CKM_SSL3_MD5_MAC
CKM_SSL3_SHA1_MAC

6.39.2 SSL mechanism parameters

◆ CK_SSL3_RANDOM_DATA

CK_SSL3_RANDOM_DATA is a structure which provides information about the random data of a client and a server in an SSL context. This structure is used by both the **CKM_SSL3_MASTER_KEY_DERIVE** and the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
typedef struct CK_SSL3_RANDOM_DATA {
    CK_BYTE_PTR    pClientRandom;
    CK_ULONG       ulClientRandomLen;
    CK_BYTE_PTR    pServerRandom;
    CK_ULONG       ulServerRandomLen;
} CK_SSL3_RANDOM_DATA;
```

The fields of the structure have the following meanings:

pClientRandom pointer to the client's random data
ulClientRandomLen length in bytes of the client's random data
pServerRandom pointer to the server's random data
ulServerRandomLen length in bytes of the server's random data

10754 ♦ **CK_SSL3_MASTER_KEY_DERIVE_PARAMS;**
 10755 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR**

10756 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** is a structure that provides the parameters to the
 10757 **CKM_SSL3_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
10758     typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {
10759         CK_SSL3_RANDOM_DATA    RandomInfo;
10760         CK_VERSION_PTR          pVersion;
10761     } CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

10762
 10763 The fields of the structure have the following meanings:

10764	RandomInfo	client's and server's random data information.
10765	pVersion	pointer to a CK_VERSION structure which receives the SSL
10766		protocol version information

10767 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
 10768 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**.

10769 ♦ **CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR**

10770 **CK_SSL3_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization vectors
 10771 after performing a C_DeriveKey function with the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It
 10772 is defined as follows:

```
10773     typedef struct CK_SSL3_KEY_MAT_OUT {
10774         CK_OBJECT_HANDLE    hClientMacSecret;
10775         CK_OBJECT_HANDLE    hServerMacSecret;
10776         CK_OBJECT_HANDLE    hClientKey;
10777         CK_OBJECT_HANDLE    hServerKey;
10778         CK_BYTE_PTR          pIVClient;
10779         CK_BYTE_PTR          pIVServer;
10780     } CK_SSL3_KEY_MAT_OUT;
```

10781
 10782 The fields of the structure have the following meanings:

10783	hClientMacSecret	key handle for the resulting Client MAC Secret key
10784	hServerMacSecret	key handle for the resulting Server MAC Secret key
10785	hClientKey	key handle for the resulting Client Secret key
10786	hServerKey	key handle for the resulting Server Secret key
10787	pIVClient	pointer to a location which receives the initialization vector (IV)
10788		created for the client (if any)
10789	pIVServer	pointer to a location which receives the initialization vector (IV)
10790		created for the server (if any)

10791 **CK_SSL3_KEY_MAT_OUT_PTR** is a pointer to a **CK_SSL3_KEY_MAT_OUT**.

10792 ♦ **CK_SSL3_KEY_MAT_PARAMS; CK_SSL3_KEY_MAT_PARAMS_PTR**

10793 **CK_SSL3_KEY_MAT_PARAMS** is a structure that provides the parameters to the
 10794 **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```

10795     typedef struct CK_SSL3_KEY_MAT_PARAMS {
10796         CK_ULONG          ulMacSizeInBits;
10797         CK_ULONG          ulKeySizeInBits;
10798         CK_ULONG          ulIVSizeInBits;
10799         CK_BBOOL          bIsExport;
10800         CK_SSL3_RANDOM_DATA RandomInfo;
10801         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
10802     } CK_SSL3_KEY_MAT_PARAMS;

```

10803

10804 The fields of the structure have the following meanings:

10805	ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the
10806		protocol handshake phase
10807	ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the
10808		protocol handshake phase
10809	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol
10810		handshake phase. If no IV is required, the length should be set to 0
10811	bIsExport	a Boolean value which indicates whether the keys have to be
10812		derived for an export version of the protocol
10813	RandomInfo	client's and server's random data information.
10814	pReturnedKeyMaterial	points to a CK_SSL3_KEY_MAT_OUT structures which receives
10815		the handles for the keys generated and the IVs

10816 **CK_SSL3_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_SSL3_KEY_MAT_PARAMS**.

10817 6.39.3 Pre-master key generation

10818 Pre-master key generation in SSL 3.0, denoted **CKM_SSL3_PRE_MASTER_KEY_GEN**, is a mechanism
 10819 which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL
 10820 version 3.0 for RSA-like cipher suites.

10821 It has one parameter, a **CK_VERSION** structure, which provides the client's SSL version number.

10822 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 10823 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
 10824 be specified in the template, or else are assigned default values.

10825 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
 10826 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
 10827 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
 10828 specify any of them.

10829 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
 10830 both indicate 48 bytes.

10831 **CKM_TLS_PRE_MASTER_KEY_GEN** has identical functionality as
 10832 **CKM_SSL3_PRE_MASTER_KEY_GEN**. It exists only for historical reasons, please use
 10833 **CKM_SSL3_PRE_MASTER_KEY_GEN** instead.

10834 6.39.4 Master key derivation

10835 Master key derivation in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE**, is a mechanism used
 10836 to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce
 10837 the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the
 10838 value of the client version, which is built into the "pre_master" key as well as a handle to the derived
 10839 "master_secret" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in Section 6.39.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template; otherwise they are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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 both indicate 48 bytes.

Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure's **pVersion** field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this structure will hold the SSL version associated with the supplied pre_master key.

Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher suites.

6.39.5 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE_DH**, is a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret key. It is used to produce the "master_secret" key used in the SSL protocol from the "pre_master" key.

It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token. This structure is defined in Section 6.39. The **pVersion** field of the structure must be set to **NULL_PTR** since the version number is not embedded in the "pre_master" key as it is for RSA-like cipher suites.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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 both indicate 48 bytes.

Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but excludes the RSA cipher suites.

6.39.6 Key and MAC derivation

Key, MAC and IV derivation in SSL 3.0, denoted **CKM_SSL3_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 6.39.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") are always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing, verification, and derivation operations.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK_SSL3_KEY_MAT_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ulIVSizeInBits* field.

All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a NULL_PTR.

10940 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the
10941 token.

10942 **6.39.7 MD5 MACing in SSL 3.0**

10943 MD5 MACing in SSL3.0, denoted **CKM_SSL3_MD5_MAC**, is a mechanism for single- and multiple-part
10944 signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This
10945 technique is very similar to the HMAC technique.

10946 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
10947 signatures produced by this mechanism.

10948 Constraints on key types and the length of input and output data are summarized in the following table:

10949 *Table 186, 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

10950 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10951 specify the supported range of generic secret key sizes, in bits.

10952 **6.39.8 SHA-1 MACing in SSL 3.0**

10953 SHA-1 MACing in SSL3.0, denoted **CKM_SSL3_SHA1_MAC**, is a mechanism for single- and multiple-
10954 part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This
10955 technique is very similar to the HMAC technique.

10956 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
10957 signatures produced by this mechanism.

10958 Constraints on key types and the length of input and output data are summarized in the following table:

10959 *Table 187, 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

10960 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10961 specify the supported range of generic secret key sizes, in bits.

10962 **6.40 TLS 1.2 Mechanisms**

10963 Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2
10964 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS
10965 PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2
10966 mechanisms contains a new value in the parameters structure to specify the hash function.

10967 This section also specifies **CKM_TLS12_MAC** which should be used in place of **CKM_TLS_PRF** to
10968 calculate the *verify_data* in the TLS "finished" message.

10969 This section also specifies **CKM_TLS_KDF** that can be used in place of **CKM_TLS_PRF** to implement
10970 key material exporters.

10971

10972 *Table 188, 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							✓

6.40.1 Definitions

Mechanisms:

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

6.40.2 TLS 1.2 mechanism parameters

◆ CK_TLS12_MASTER_KEY_DERIVE_PARAMS; CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR

CK_TLS12_MASTER_KEY_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_TLS12_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_TLS12_MASTER_KEY_DERIVE_PARAMS {
    CK_SSL3_RANDOM_DATA RandomInfo;
    CK_VERSION_PTR pVersion;
    CK_MECHANISM_TYPE prfHashMechanism;
} CK_TLS12_MASTER_KEY_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

RandomInfo	client's and server's random data information.
pVersion	pointer to a CK_VERSION structure which receives the SSL protocol version information
prfHashMechanism	base hash used in the underlying TLS1.2 PRF operation used to derive the master key.

CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR is a pointer to a **CK_TLS12_MASTER_KEY_DERIVE_PARAMS**.

11002 ◆ **CK_TLS12_KEY_MAT_PARAMS; CK_TLS12_KEY_MAT_PARAMS_PTR**

11003 **CK_TLS12_KEY_MAT_PARAMS** is a structure that provides the parameters to the
11004 **CKM_TLS12_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
11005         typedef struct CK_TLS12_KEY_MAT_PARAMS {  
11006             CK_ULONG ulMacSizeInBits;  
11007             CK_ULONG ulKeySizeInBits;  
11008             CK_ULONG ulIVSizeInBits;  
11009             CK_BBOOL bIsExport;  
11010             CK_SSL3_RANDOM_DATA RandomInfo;  
11011             CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
11012             CK_MECHANISM_TYPE prfHashMechanism;  
11013         } CK_TLS12_KEY_MAT_PARAMS;
```

11014

11015 The fields of the structure have the following meanings:

11016	ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the
11017		protocol handshake phase. If no MAC key is required, the length
11018		should be set to 0.
11019	ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the
11020		protocol handshake phase
11021	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol
11022		handshake phase. If no IV is required, the length should be set to 0
11023	bIsExport	must be set to CK_FALSE because export cipher suites must not be
11024		used in TLS 1.1 and later.
11025	RandomInfo	client's and server's random data information.
11026	pReturnedKeyMaterial	points to a CK_SSL3_KEY_MAT_OUT structures which receives
11027		the handles for the keys generated and the IVs
11028	prfHashMechanism	base hash used in the underlying TLS1.2 PRF operation used to
11029		derive the master key.

11030 **CK_TLS12_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_TLS12_KEY_MAT_PARAMS**.

11031 ◆ **CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR**

11032 **CK_TLS_KDF_PARAMS** is a structure that provides the parameters to the CKM_TLS_KDF mechanism.
11033 It is defined as follows:

```
11034         typedef struct CK_TLS_KDF_PARAMS {  
11035             CK_MECHANISM_TYPE prfMechanism;  
11036             CK_BYTE_PTR pLabel;  
11037             CK_ULONG ulLabelLength;  
11038             CK_SSL3_RANDOM_DATA RandomInfo;  
11039             CK_BYTE_PTR pContextData;  
11040             CK_ULONG ulContextDataLength;  
11041         } CK_TLS_KDF_PARAMS;
```

11042

11043 The fields of the structure have the following meanings:

11044	prfMechanism	the hash mechanism used in the TLS1.2 PRF construct or
11045		CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.

11046	pLabel	a pointer to the label for this key derivation
11047	ulLabelLength	length of the label in bytes
11048	RandomInfo	the random data for the key derivation
11049	pContextData	a pointer to the context data for this key derivation. NULL_PTR if not
11050		present
11051	ulContextDataLength	length of the context data in bytes. 0 if not present.

11052 **CK_TLS_KDF_PARAMS_PTR** is a pointer to a **CK_TLS_KDF_PARAMS**.

11053 ♦ **CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR**

11054 **CK_TLS_MAC_PARAMS** is a structure that provides the parameters to the **CKM_TLS_MAC**
 11055 mechanism. It is defined as follows:

```

11056     typedef struct CK_TLS_MAC_PARAMS {
11057         CK_MECHANISM_TYPE prfHashMechanism;
11058         CK_ULONG ulMacLength;
11059         CK_ULONG ulServerOrClient;
11060     } CK_TLS_MAC_PARAMS;
11061 
```

11062 The fields of the structure have the following meanings:

11063	prfHashMechanism	the hash mechanism used in the TLS12 PRF construct or
11064		CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
11065	ulMacLength	the length of the MAC tag required or offered. Always 12 octets in
11066		TLS 1.0 and 1.1. Generally 12 octets, but may be negotiated to a
11067		longer value in TLS1.2.
11068	ulServerOrClient	1 to use the label "server finished", 2 to use the label "client
11069		finished". All other values are invalid.

11070 **CK_TLS_MAC_PARAMS_PTR** is a pointer to a **CK_TLS_MAC_PARAMS**.

11071

11072 ♦ **CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR**

11073 **CK_TLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_TLS_PRF**
 11074 mechanism. It is defined as follows:

```

11075     typedef struct CK_TLS_PRF_PARAMS {
11076         CK_BYTE_PTR pSeed;
11077         CK_ULONG ulSeedLen;
11078         CK_BYTE_PTR pLabel;
11079         CK_ULONG ulLabelLen;
11080         CK_BYTE_PTR pOutput;
11081         CK_ULONG_PTR pulOutputLen;
11082     } CK_TLS_PRF_PARAMS;
11083 
```

11084 The fields of the structure have the following meanings:

11085	pSeed	<i>pointer to the input seed</i>
11086	ulSeedLen	<i>length in bytes of the input seed</i>
11087	pLabel	<i>pointer to the identifying label</i>

11131 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 11132 **CKA_SENSITIVE** attribute.

- 11133 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
 11134 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 11135 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 11136 value from its **CKA_EXTRACTABLE** attribute.

11137 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
 11138 both indicate 48 bytes.

11139 Note that the **CK_VERSION** structure pointed to by the **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**
 11140 structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns,
 11141 this structure will hold the SSL version associated with the supplied pre_master key.

11142 Note that this mechanism is only useable for cipher suites that use a 48-byte "pre_master" secret with an
 11143 embedded version number. This includes the RSA cipher suites, but excludes the Diffie-Hellman cipher
 11144 suites.

11145 6.40.5 Master key derivation for Diffie-Hellman

11146 Master key derivation for Diffie-Hellman in TLS 1.0, denoted **CKM_TLS_MASTER_KEY_DERIVE_DH**, is
 11147 a mechanism used to derive one 48-byte generic secret key from another arbitrary length generic secret
 11148 key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key.

11149 It has a parameter, a **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the
 11150 passing of random data to the token. This structure is defined in Section 6.39. The *pVersion* field of the
 11151 structure must be set to NULL_PTR since the version number is not embedded in the "pre_master" key
 11152 as it is for RSA-like cipher suites.

11153 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 11154 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
 11155 be specified in the template, or else are assigned default values.

11156 The mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of
 11157 **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS12_KDF** and
 11158 **CKM_TLS12_MAC**.

11159 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
 11160 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
 11161 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
 11162 specify any of them.

11163 This mechanism has the following rules about key sensitivity and extractability:

- 11164 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
 11165 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
 11166 default value.
- 11167 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
 11168 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
 11169 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 11170 **CKA_SENSITIVE** attribute.
- 11171 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
 11172 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 11173 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 11174 value from its **CKA_EXTRACTABLE** attribute.

11175 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
 11176 both indicate 48 bytes.

11177 Note that this mechanism is only useable for cipher suites that do not use a fixed length 48-byte
 11178 "pre_master" secret with an embedded version number. This includes the Diffie-Hellman cipher suites, but
 11179 excludes the RSA cipher suites.

6.40.6 Key and MAC derivation

Key, MAC and IV derivation in TLS 1.0, denoted **CKM_TLS_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a "CipherSuite" from the "master_secret" key and random data. This mechanism returns the key handles for the keys generated in the process, as well as the IVs created.

It has a parameter, a **CK_SSL3_KEY_MAT_PARAMS** structure, which allows for the passing of random data as well as the characteristic of the cryptographic material for the given CipherSuite and a pointer to a structure which receives the handles and IVs which were generated. This structure is defined in Section 6.39.

This mechanism contributes to the creation of four distinct keys on the token and returns two IVs (if IVs are requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The two MACing keys ("client_write_MAC_secret" and "server_write_MAC_secret") (if present) are always given a type of **CKK_GENERIC_SECRET**. They are flagged as valid for signing and verification.

The other two keys ("client_write_key" and "server_write_key") are typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, they are flagged as valid for encryption, decryption, and derivation operations.

For **CKM_TLS12_KEY_AND_MAC_DERIVE**, IVs will be generated and returned if the *ulIVSizeInBits* field of the **CK_SSL3_KEY_MAT_PARAMS** field has a nonzero value. If they are generated, their length in bits will agree with the value in the *ulIVSizeInBits* field.

Note Well: **CKM_TLS12_KEY_AND_MAC_DERIVE** produces both private (key) and public (IV) data. It is possible to "leak" private data by the simple expedient of decreasing the length of private data requested. E.g. Setting *ulMacSizeInBits* and *ulKeySizeInBits* to 0 (or other lengths less than the key size) will result in the private key data being placed in the destination designated for the IV's. Repeated calls with the same master key and same RandomInfo but with differing lengths for the private key material will result in different data being leaked.<

All four keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template provided to **C_DeriveKey** may not specify values for any of these attributes which differ from those held by the base key.

Note that the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the four key handle fields in the **CK_SSL3_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffers pointed to by the **CK_SSL3_KEY_MAT_OUT** structure's *pIVClient* and *pIVServer* fields will have IVs returned in them (if IVs are requested by the caller). Therefore, these two fields must point to buffers with sufficient space to hold any IVs that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_SSL3_KEY_MAT_OUT** structure pointed to by the **CK_SSL3_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a **NULL_PTR**.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the token.

6.40.7 CKM_TLS12_KEY_SAFE_DERIVE

CKM_TLS12_KEY_SAFE_DERIVE is identical to **CKM_TLS12_KEY_AND_MAC_DERIVE** except that it shall never produce IV data, and the *ulIVSizeInBits* field of **CK_TLS12_KEY_MAT_PARAMS** is ignored

11229 and treated as 0. All of the other conditions and behavior described for
11230 CKM_TLS12_KEY_AND_MAC_DERIVE, with the exception of the black box warning, apply to this
11231 mechanism.
11232 CKM_TLS12_KEY_SAFE_DERIVE is provided as a separate mechanism to allow a client to control the
11233 export of IV material (and possible leaking of key material) through the use of the
11234 CKA_ALLOWED_MECHANISMS key attribute.

11235 6.40.8 Generic Key Derivation using the TLS PRF

11236 **CKM_TLS_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
11237 function to produce additional key material for protocols that want to leverage the TLS key negotiation
11238 mechanism. **CKM_TLS_KDF** has a parameter of **CK_TLS_KDF_PARAMS**. If the protocol using this
11239 mechanism does not use context information, the *pContextData* field shall be set to NULL_PTR and the
11240 *ulContextDataLength* field shall be set to 0.

11241 To use this mechanism with TLS1.0 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in
11242 place of a hash mechanism. Note: Although **CKM_TLS_PRF** is deprecated as a mechanism for
11243 C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the
11244 TLS1.0/1.1 Pseudo-random function.

11245 This mechanism can be used to derive multiple keys (e.g. similar to
11246 **CKM_TLS12_KEY_AND_MAC_DERIVE**) by first deriving the key stream as a **CKK_GENERIC_SECRET**
11247 of the necessary length and doing subsequent derives against that derived key using the
11248 **CKM_EXTRACT_KEY_FROM_KEY** mechanism to split the key stream into the actual operational keys.

11249 The mechanism should not be used with the labels defined for use with TLS, but the token does not
11250 enforce this behavior.

11251 This mechanism has the following rules about key sensitivity and extractability:

- 11252 • If the original key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
11253 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from the
11254 original key.
- 11255 • Similarly, if the original key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
11256 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
11257 supplied template or from the original key.
- 11258 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the original
11259 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 11260 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
11261 the original key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

11262 6.40.9 Generic Key Derivation using the TLS12 PRF

11263 **CKM_TLS12_KDF** is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF
11264 function to produce additional key material for protocols that want to leverage the TLS key negotiation
11265 mechanism. **CKM_TLS12_KDF** has a parameter of **CK_TLS_KDF_PARAMS**. If the protocol using this
11266 mechanism does not use context information, the *pContextData* field shall be set to NULL_PTR and the
11267 *ulContextDataLength* field shall be set to 0.

11268 To use this mechanism with TLS1.0 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in
11269 place of a hash mechanism. Note: Although **CKM_TLS_PRF** is deprecated as a mechanism for
11270 C_DeriveKey, the manifest value is retained for use with this mechanism to indicate the use of the
11271 TLS1.0/1.1 Pseudo-random function.

11272 This mechanism can be used to derive multiple keys (e.g. similar to
11273 **CKM_TLS12_KEY_AND_MAC_DERIVE**) by first deriving the key stream as a **CKK_GENERIC_SECRET**
11274 of the necessary length and doing subsequent derives against that derived key stream using the
11275 **CKM_EXTRACT_KEY_FROM_KEY** mechanism to split the key stream into the actual operational keys.

- 11276 The mechanism should not be used with the labels defined for use with TLS, but the token does not
 11277 enforce this behavior.
- 11278 This mechanism has the following rules about key sensitivity and extractability:
- 11279 • If the original key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
 11280 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from the
 11281 original key.
 - 11282 • Similarly, if the original key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 11283 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
 11284 supplied template or from the original key.
 - 11285 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the original
 11286 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
 - 11287 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 11288 the original key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

11289 **6.41 WTLS**

- 11290 Details can be found in [WTLS].
- 11291 When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue
 11292 that there would be no need to have distinct handling of the client and server side of the handshake.
 11293 However, since in WTLS the server and client use different sequence numbers, there could be instances
 11294 (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and
 11295 server side differ, and hence this motivates the introduced split.

11296

11297 *Table 190, WTLS Mechanisms vs. Functions*

Mechanism	Functions						
	Encry pt & Decry pt	Sign & Verif y	SR & VR 1	Dige st	Ge n. Key / Key Pair	Wrap & Unwra p	Deriv e
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							✓

11298 **6.41.1 Definitions**

- 11299 Mechanisms:
- 11300 CKM_WTLS_PRE_MASTER_KEY_GEN
 - 11301 CKM_WTLS_MASTER_KEY_DERIVE
 - 11302 CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC

11303 CKM_WTLS_PRF
 11304 CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
 11305 CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE

11306 6.41.2 WTLS mechanism parameters

11307 ♦ CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR

11308 **CK_WTLS_RANDOM_DATA** is a structure, which provides information about the random data of a client
 11309 and a server in a WTLS context. This structure is used by the **CKM_WTLS_MASTER_KEY_DERIVE**
 11310 mechanism. It is defined as follows:

```
11311 typedef struct CK_WTLS_RANDOM_DATA {
11312     CK_BYTE_PTR pClientRandom;
11313     CK_ULONG     ulClientRandomLen;
11314     CK_BYTE_PTR pServerRandom;
11315     CK_ULONG     ulServerRandomLen;
11316 } CK_WTLS_RANDOM_DATA;
11317
```

11318 The fields of the structure have the following meanings:

11319	pClientRandom	pointer to the client's random data
11320	pClientRandomLen	length in bytes of the client's random data
11321	pServerRandom	pointer to the server's random data
11322	ulServerRandomLen	length in bytes of the server's random data

11323 **CK_WTLS_RANDOM_DATA_PTR** is a pointer to a **CK_WTLS_RANDOM_DATA**.

11324 ♦ CK_WTLS_MASTER_KEY_DERIVE_PARAMS; 11325 CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR

11326 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** is a structure, which provides the parameters to the
 11327 **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
11328 typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {
11329     CK_MECHANISM_TYPE DigestMechanism;
11330     CK_WTLS_RANDOM_DATA RandomInfo;
11331     CK_BYTE_PTR pVersion;
11332 } CK_WTLS_MASTER_KEY_DERIVE_PARAMS;
11333
```

11334 The fields of the structure have the following meanings:

11335	DigestMechanism	the mechanism type of the digest mechanism to be used (possible types can be found in [WTLS])
11337	RandomInfo	Client's and server's random data information
11338	pVersion	pointer to a CK_BYTE which receives the WTLS protocol version information

11340 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
 11341 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS**.

11342 ♦ **CK_WTLS_PRF_PARAMS; CK_WTLS_PRF_PARAMS_PTR**

11343 **CK_WTLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_WTLS_PRF**
 11344 mechanism. It is defined as follows:

```
11345     typedef struct CK_WTLS_PRF_PARAMS {
11346         CK_MECHANISM_TYPE DigestMechanism;
11347         CK_BYTE_PTR      pSeed;
11348         CK_ULONG         ulSeedLen;
11349         CK_BYTE_PTR      pLabel;
11350         CK_ULONG         ulLabelLen;
11351         CK_BYTE_PTR      pOutput;
11352         CK_ULONG_PTR     pulOutputLen;
11353     } CK_WTLS_PRF_PARAMS;
```

11354

11355 The fields of the structure have the following meanings:

11356	Digest Mechanism	the mechanism type of the digest mechanism to be used (possible
11357		types can be found in [WTLS])
11358	pSeed	pointer to the input seed
11359	ulSeedLen	length in bytes of the input seed
11360	pLabel	pointer to the identifying label
11361	ulLabelLen	length in bytes of the identifying label
11362	pOutput	pointer receiving the output of the operation
11363	pulOutputLen	pointer to the length in bytes that the output to be created shall
11364		have, has to hold the desired length as input and will receive the
11365		calculated length as output

11366 **CK_WTLS_PRF_PARAMS_PTR** is a pointer to a **CK_WTLS_PRF_PARAMS**.

11367 ♦ **CK_WTLS_KEY_MAT_OUT; CK_WTLS_KEY_MAT_OUT_PTR**

11368 **CK_WTLS_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization
 11369 vectors after performing a **C_DeriveKey** function with the
 11370 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** or with the
 11371 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```
11372     typedef struct CK_WTLS_KEY_MAT_OUT {
11373         CK_OBJECT_HANDLE hMacSecret;
11374         CK_OBJECT_HANDLE hKey;
11375         CK_BYTE_PTR      pIV;
11376     } CK_WTLS_KEY_MAT_OUT;
```

11377

11378 The fields of the structure have the following meanings:

11379	hMacSecret	Key handle for the resulting MAC secret key
11380	hKey	Key handle for the resulting secret key
11381	pIV	Pointer to a location which receives the initialization vector (IV)
11382		created (if any)

11383 **CK_WTLS_KEY_MAT_OUT_PTR** is a pointer to a **CK_WTLS_KEY_MAT_OUT**.

11384 ♦ **CK_WTLS_KEY_MAT_PARAMS; CK_WTLS_KEY_MAT_PARAMS_PTR**

11385 **CK_WTLS_KEY_MAT_PARAMS** is a structure that provides the parameters to the
11386 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** and the
11387 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```
11388     typedef struct CK_WTLS_KEY_MAT_PARAMS {  
11389         CK_MECHANISM_TYPE      DigestMechanism;  
11390         CK_ULONG               ulMacSizeInBits;  
11391         CK_ULONG               ulKeySizeInBits;  
11392         CK_ULONG               ulIVSizeInBits;  
11393         CK_ULONG               ulSequenceNumber;  
11394         CK_BBOOL               bIsExport;  
11395         CK_WTLS_RANDOM_DATA    RandomInfo;  
11396         CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;  
11397     } CK_WTLS_KEY_MAT_PARAMS;
```

11398

11399 The fields of the structure have the following meanings:

11400	Digest Mechanism	the mechanism type of the digest mechanism to be used (possible
11401		types can be found in [WTLS])
11402	ulMaxSizeInBits	the length (in bits) of the MACing key agreed upon during the
11403		protocol handshake phase
11404	ulKeySizeInBits	the length (in bits) of the secret key agreed upon during the
11405		handshake phase
11406	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the handshake
11407		phase. If no IV is required, the length should be set to 0.
11408	ulSequenceNumber	the current sequence number used for records sent by the client
11409		and server respectively
11410	bIsExport	a boolean value which indicates whether the keys have to be
11411		derives for an export version of the protocol. If this value is true
11412		(i.e., the keys are exportable) then ulKeySizeInBits is the length of
11413		the key in bits before expansion. The length of the key after
11414		expansion is determined by the information found in the template
11415		sent along with this mechanism during a C_DeriveKey function call
11416		(either the CKA_KEY_TYPE or the CKA_VALUE_LEN attribute).
11417	RandomInfo	client's and server's random data information
11418	pReturnedKeyMaterial	points to a CK_WTLS_KEY_MAT_OUT structure which receives
11419		the handles for the keys generated and the IV

11420 **CK_WTLS_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_WTLS_KEY_MAT_PARAMS**.

11421 **6.41.3 Pre master secret key generation for RSA key exchange suite**

11422 Pre master secret key generation for the RSA key exchange suite in WTLS denoted
11423 **CKM_WTLS_PRE_MASTER_KEY_GEN**, is a mechanism, which generates a variable length secret key.
11424 It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This
11425 mechanism returns a handle to the pre master secret key.

11426 It has one parameter, a **CK_BYTE**, which provides the client's WTLS version.

11427 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE** and **CKA_VALUE** attributes to the new
11428 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
11429 be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute indicates the length of the pre master secret key.

For this mechanism, the **ulMinKeySize** field of the **CK_MECHANISM_INFO** structure shall indicate 20 bytes.

6.41.4 Master secret key derivation

Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client version, which is built into the pre master secret key as well as a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used as well as the passing of random data to the token as well as the returning of the protocol version number which is part of the pre master secret key.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN** attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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 both indicate 20 bytes.

Note that the **CK_BYTE** pointed to by the **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure's *pVersion* field will be modified by the **C_DeriveKey** call. In particular, when the call returns, this byte will hold the WTLS version associated with the supplied pre master secret key.

Note that this mechanism is only useable for key exchange suites that use a 20-byte pre master secret key with an embedded version number. This includes the RSA key exchange suites, but excludes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites.

6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography

Master secret derivation for Diffie-Hellman and Elliptic Curve Cryptography in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC**, is a mechanism used to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master secret key used in WTLS from the pre master secret key. This mechanism returns a handle to the derived master secret key.

It has a parameter, a **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used as well as random data to the token. The *pVersion* field of the structure must be set to **NULL_PTR** since the version number is not embedded in the pre master secret key as it is for RSA-like key exchange suites.

11479 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11480 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
11481 be specified in the template, or else are assigned default values.

11482 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
11483 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
11484 attribute has value 20. However, since these facts are all implicit in the mechanism, there is no need to
11485 specify any of them.

11486 This mechanism has the following rules about key sensitivity and extractability:

11487 The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both be
11488 specified to be either **CK_TRUE** or **CK_FALSE**. If omitted, these attributes each take on some default
11489 value.

11490 If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_FALSE**, then the derived key will
11491 as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to **CK_TRUE**, then the derived
11492 key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its **CKA_SENSITIVE** attribute.

11493 Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to **CK_FALSE**, then the
11494 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to **CK_TRUE**,
11495 then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite* value from its
11496 **CKA_EXTRACTABLE** attribute.

11497 For this mechanism, the **ulMinKeySize** and **ulMaxKeySize** fields of the **CK_MECHANISM_INFO** structure
11498 both indicate 20 bytes.

11499 Note that this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte
11500 pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic
11501 Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

11502 6.41.6 WTLS PRF (pseudorandom function)

11503 PRF (pseudo random function) in WTLS, denoted **CKM_WTLS_PRF**, is a mechanism used to produce a
11504 securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

11505 It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which allows for passing the mechanism type
11506 of the digest mechanism to be used, the passing of the input seed and its length, the passing of an
11507 identifying label and its length and the passing of the length of the output to the token and for receiving
11508 the output.

11509 This mechanism produces securely generated pseudo-random output of the length specified in the
11510 parameter.

11511 This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template
11512 sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a
11513 **NULL_PTR**. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result
11514 of a successful completion. However, since the **CKM_WTLS_PRF** mechanism returns the requested
11515 number of output bytes in the **CK_WTLS_PRF_PARAMS** structure specified as the mechanism
11516 parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a **NULL_PTR**.

11517 If a call to **C_DeriveKey** with this mechanism fails, then no output will be generated.

11518 6.41.7 Server Key and MAC derivation

11519 Server key, MAC and IV derivation in WTLS, denoted
11520 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate
11521 cryptographic keying material used by a cipher suite from the master secret key and random data. This
11522 mechanism returns the key handles for the keys generated in the process, as well as the IV created.

11523 It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the
11524 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic
11525 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which
11526 were generated.

11527 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested
 11528 by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

11529 The MACing key (server write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is
 11530 flagged as valid for signing, verification and derivation operations.

11531 The other key (server write key) is typed according to information found in the template sent along with
 11532 this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption,
 11533 decryption, and derivation operations.

11534 An IV (server write IV) will be generated and returned if the *ulIVSizeInBits* field of the
 11535 **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree
 11536 with the value in the *ulIVSizeInBits* field

11537 Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
 11538 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
 11539 provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by
 11540 the base key.

11541 Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS**
 11542 structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key
 11543 handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-
 11544 created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will
 11545 have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a
 11546 buffer with sufficient space to hold any IV that will be returned.

11547 This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information.
 11548 For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a
 11549 successful completion. However, since the **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**
 11550 mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the
 11551 **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey*
 11552 passed to **C_DeriveKey** is unnecessary, and should be a **NULL_PTR**.

11553 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

11554 6.41.8 Client key and MAC derivation

11555 Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**,
 11556 is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite from
 11557 the master secret key and random data. This mechanism returns the key handles for the keys generated
 11558 in the process, as well as the IV created.

11559 It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the
 11560 mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic
 11561 material for the given cipher suite, and a pointer to a structure which receives the handles and IV which
 11562 were generated.

11563 This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested
 11564 by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

11565 The MACing key (client write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is
 11566 flagged as valid for signing, verification and derivation operations.

11567 The other key (client write key) is typed according to information found in the template sent along with this
 11568 mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption,
 11569 decryption, and derivation operations.

11570 An IV (client write IV) will be generated and returned if the *ulIVSizeInBits* field of the
 11571 **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree
 11572 with the value in the *ulIVSizeInBits* field

11573 Both keys inherit the values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**,
 11574 **CKA_EXTRACTABLE**, and **CKA_NEVER_EXTRACTABLE** attributes from the base key. The template
 11575 provided to **C_DeriveKey** may not specify values for any of these attributes that differ from those held by
 11576 the base key.

Note that the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure's *pReturnedKeyMaterial* field will be modified by the **C_DeriveKey** call. In particular, the two key handle fields in the **CK_WTLS_KEY_MAT_OUT** structure will be modified to hold handles to the newly-created keys; in addition, the buffer pointed to by the **CK_WTLS_KEY_MAT_OUT** structure's *pIV* field will have the IV returned in them (if an IV is requested by the caller). Therefore, this field must point to a buffer with sufficient space to hold any IV that will be returned.

This mechanism departs from the other key derivation mechanisms in Cryptoki in its returned information. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism returns all of its key handles in the **CK_WTLS_KEY_MAT_OUT** structure pointed to by the **CK_WTLS_KEY_MAT_PARAMS** structure specified as the mechanism parameter, the parameter *phKey* passed to **C_DeriveKey** is unnecessary, and should be a **NULL_PTR**.

If a call to **C_DeriveKey** with this mechanism fails, then *none* of the two keys will be created.

6.42 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 accepts 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_PRF_DATA_PARAM** array that is part of the mechanism parameter.

Table 191, 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.

6.42.1 Definitions

Mechanisms:

CKM_SP800_108_COUNTER_KDF

CKM_SP800_108_FEEDBACK_KDF

11615 CKM_SP800_108_DOUBLE_PIPELINE_KDF

11616

11617 Data Field Types:

11618 CK_SP800_108_ITERATION_VARIABLE

11619 CK_SP800_108_COUNTER

11620 CK_SP800_108_DKM_LENGTH

11621 CK_SP800_108_BYTE_ARRAY

11622

11623 DKM Length Methods:

11624 CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS

11625 CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS

11626 **6.42.2 Mechanism Parameters**

11627 **◆ CK_SP800_108_PRF_TYPE**

11628 The **CK_SP800_108_PRF_TYPE** field of the mechanism parameter is used to specify the type of PRF

11629 that is to be used. It is defined as follows:

11630 `typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;`

11631 The **CK_SP800_108_PRF_TYPE** field reuses the existing mechanisms definitions. The following table

11632 lists the supported PRF types:

11633 *Table 192, 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_DES3_CMAC
CKM_AES_CMAC

11634

11635 **◆ CK_PRF_DATA_TYPE**

11636 Each mechanism parameter contains an array of **CK_PRF_DATA_PARAM** structures. The

11637 **CK_PRF_DATA_PARAM** structure contains **CK_PRF_DATA_TYPE** field. The **CK_PRF_DATA_TYPE**

11638 field is used to identify the type of data identified by each **CK_PRF_DATA_PARAM** element in the array.

11639 Depending on the type of KDF used, some data field types are mandatory, some data field types are

11640 optional and some data field types are not allowed. These requirements are defined on a per-mechanism

11641 basis in the sections below. The **CK_PRF_DATA_TYPE** is defined as follows:

11642 `typedef CK_ULONG CK_PRF_DATA_TYPE;`

11643 The following table lists all of the supported data field types:

11644 Table 193, 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_LENGTH_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.

11645

11646 ♦ **CK_PRF_DATA_PARAM**

11647 **CK_PRF_DATA_PARAM** is used to define a segment of input for the PRF. Each mechanism parameter
11648 supports an array of **CK_PRF_DATA_PARAM** structures. The **CK_PRF_DATA_PARAM** is defined as
11649 follows:

```
11650     typedef struct CK_PRF_DATA_PARAM
11651     {
11652         CK_PRF_DATA_TYPE      type;
11653         CK_VOID_PTR           pValue;
11654         CK_ULONG              ulValueLen;
11655     } CK_PRF_DATA_PARAM;
11656
11657     typedef CK_PRF_DATA_PARAM CK_PTR CK_PRF_DATA_PARAM_PTR
```

11658

11659 The fields of the **CK_PRF_DATA_PARAM** structure have the following meaning:

11660 type defines the type of data pointed to by pValue

11661 pValue pointer to the data defined by type

11662 ulValueLen size of the data pointed to by pValue

11663 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to

11664 CK_SP800_108_ITERATION_VARIABLE, then *pValue* must be set the appropriate value for the KDF's

11665 iteration variable type. For the Counter Mode KDF, *pValue* must be assigned a valid

11666 CK_SP800_108_COUNTER_FORMAT_PTR and *ulValueLen* must be set to

11667 sizeof(CK_SP800_108_COUNTER_FORMAT). For all other KDF types, *pValue must be set to*

11668 NULL_PTR and *ulValueLen* must be set to 0.

11669

11670 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_COUNTER, then

11671 *pValue* must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and *ulValueLen* must be

11672 set to sizeof(CK_SP800_108_COUNTER_FORMAT).

11673

11674 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_DKM_LENGTH then

11675 *pValue* must be assigned a valid CK_SP800_108_DKM_LENGTH_FORMAT_PTR and *ulValueLen* must

11676 be set to sizeof(CK_SP800_108_DKM_LENGTH_FORMAT).

11677

11678 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_BYTE_ARRAY, then

11679 *pValue* must be assigned a valid CK_BYTE_PTR value and *ulValueLen* must be set to a non-zero length.

11680 **◆ CK_SP800_108_COUNTER_FORMAT**

11681 **CK_SP800_108_COUNTER_FORMAT** is used to define the encoding format for a counter value. The

11682 **CK_SP800_108_COUNTER_FORMAT** is defined as follows:

```
11683     typedef struct CK_SP800_108_COUNTER_FORMAT
11684     {
11685         CK_BBOOL      bLittleEndian;
11686         CK_ULONG      ulWidthInBits;
11687     } CK_SP800_108_COUNTER_FORMAT;
11688
11689     typedef CK_SP800_108_COUNTER_FORMAT CK_PTR
11690     CK_SP800_108_COUNTER_FORMAT_PTR
```

11691

11692 The fields of the CK_SP800_108_COUNTER_FORMAT structure have the following meaning:

11693 bLittleEndian defines if the counter should be represented in Big Endian or Little

11694 Endian format

11695 ulWidthInBits defines the number of bits used to represent the counter value

11696 **◆ CK_SP800_108_DKM_LENGTH_METHOD**

11697 **CK_SP800_108_DKM_LENGTH_METHOD** is used to define how the DKM length value is calculated.

11698 The **CK_SP800_108_DKM_LENGTH_METHOD** type is defined as follows:

```
11699     typedef CK_ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

11700 The following table lists all of the supported DKM Length Methods:

11701 *Table 194, 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.

11702

11703 **◆ CK_SP800_108_DKM_LENGTH_FORMAT**

11704 **CK_SP800_108_DKM_LENGTH_FORMAT** is used to define the encoding format for the DKM length

11705 value. The **CK_SP800_108_DKM_LENGTH_FORMAT** is defined as follows:

```
11706     typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
```



```

11707     {
11708         CK_SP800_108_DKM_LENGTH_METHOD    dkmLengthMethod;
11709         CK_BBOOL                           bLittleEndian;
11710         CK_ULONG                           ulWidthInBits;
11711     } CK_SP800_108_DKM_LENGTH_FORMAT;
11712
11713     typedef CK_SP800_108_DKM_LENGTH_FORMAT CK_PTR
11714     CK_SP800_108_DKM_LENGTH_FORMAT_PTR
11715

```

The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:

11716	dkmLengthMethod	defines the method used to calculate the DKM length value
11718	bLittleEndian	defines if the DKM length value should be represented in Big
11719		Endian or Little Endian format
11720	ulWidthInBits	defines the number of bits used to represent the DKM length value

11721 ♦ CK_DERIVED_KEY

11722 **CK_DERIVED_KEY** is used to define an additional key to be derived as well as provide a
11723 CK_OBJECT_HANDLE_PTR to receive the handle for the derived keys. The **CK_DERIVED_KEY** is
11724 defined as follows:

```

11725     typedef struct CK_DERIVED_KEY
11726     {
11727         CK_ATTRIBUTE_PTR    pTemplate;
11728         CK_ULONG             ulAttributeCount;
11729         CK_OBJECT_HANDLE_PTR phKey;
11730     } CK_DERIVED_KEY;
11731
11732     typedef CK_DERIVED_KEY CK_PTR CK_DERIVED_KEY_PTR
11733

```

The fields of the CK_DERIVED_KEY structure have the following meaning:

11734	pTemplate	pointer to a template that defines a key to derive
11736	ulAttributeCount	number of attributes in the template pointed to by pTemplate
11737	phKey	pointer to receive the handle for a derived key

11738 ♦ CK_SP800_108_KDF_PARAMS, CK_SP800_108_KDF_PARAMS_PTR

11739 **CK_SP800_108_KDF_PARAMS** is a structure that provides the parameters for the
11740 CKM_SP800_108_COUNTER_KDF and CKM_SP800_108_DOUBLE_PIPELINE_KDF mechanisms.

```

11741
11742     typedef struct CK_SP800_108_KDF_PARAMS
11743     {
11744         CK_SP800_108_PRF_TYPE    prfType;
11745         CK_ULONG                 ulNumberOfDataParams;
11746         CK_PRF_DATA_PARAM_PTR    pDataParams;
11747         CK_ULONG                 ulAdditionalDerivedKeys;
11748         CK_DERIVED_KEY_PTR       pAdditionalDerivedKeys;
11749     } CK_SP800_108_KDF_PARAMS;

```

```

11750
11751     typedef CK_SP800_108_KDF_PARAMS CK_PTR
11752     CK_SP800_108_KDF_PARAMS_PTR;
11753
11754 The fields of the CK_SP800_108_KDF_PARAMS structure have the following meaning:
11755     prfType      type of PRF
11756     ulNumberOfDataParams  number of elements in the array pointed to by pDataParams
11757     pDataParams   an array of CK_PRF_DATA_PARAM structures. The array defines
11758                   input parameters that are used to construct the “data” input to the
11759                   PRF.
11760     ulAdditionalDerivedKeys  number of additional keys that will be derived and the number of
11761                             elements in the array pointed to by pAdditionalDerivedKeys. If
11762                             pAdditionalDerivedKeys is set to NULL_PTR, this parameter must
11763                             be set to 0.
11764     pAdditionalDerivedKeys   an array of CK_DERIVED_KEY structures. If
11765                             ulAdditionalDerivedKeys is set to 0, this parameter must be set to
11766                             NULL_PTR

```

11767 ♦ **CK_SP800_108_FEEDBACK_KDF_PARAMS,**
11768 **CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR**

11769 The **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure provides the parameters for the
11770 CKM_SP800_108_FEEDBACK_KDF mechanism. It is defined as follows:

```

11771     typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
11772     {
11773         CK_SP800_108_PRF_TYPE    prfType;
11774         CK_ULONG                 ulNumberOfDataParams;
11775         CK_PRF_DATA_PARAM_PTR    pDataParams;
11776         CK_ULONG                 ulIVLen;
11777         CK_BYTE_PTR              pIV;
11778         CK_ULONG                 ulAdditionalDerivedKeys;
11779         CK_DERIVED_KEY_PTR       pAdditionalDerivedKeys;
11780     } CK_SP800_108_FEEDBACK_KDF_PARAMS;
11781
11782     typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR
11783     CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;
11784

```

11785 The fields of the **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure have the following meaning:

```

11786     prfType      type of PRF
11787     ulNumberOfDataParams  number of elements in the array pointed to by pDataParams
11788     pDataParams   an array of CK_PRF_DATA_PARAM structures. The array defines
11789                   input parameters that are used to construct the “data” input to the
11790                   PRF.
11791     ulIVLen       the length in bytes of the IV. If pIV is set to NULL_PTR, this
11792                   parameter must be set to 0.
11793     pIV           an array of bytes to be used as the IV for the feedback mode KDF.
11794                   This parameter is optional and can be set to NULL_PTR. If ulIVLen
11795                   is set to 0, this parameter must be set to NULL_PTR.

```


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 $K(i-1)$ in section 5.2 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>

11822

11823 SP800-108 limits the amount of derived keying material that can be produced by a Feedback Mode KDF
11824 by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be
11825 produced is $(2^{32}-1)h$, where "h" is the length in bits of the output of the selected PRF.

11826 6.42.5 Double Pipeline Mode KDF

11827 The SP800-108 Double Pipeline Mode KDF mechanism, denoted
11828 **CKM_SP800_108_DOUBLE_PIPELINE_KDF**, represents the KDF defined SP800-108 section 5.3.
11829 **CKM_SP800_108_DOUBLE_PIPELINE_KDF** is a mechanism for deriving one or more symmetric keys
11830 from a symmetric base key.

11831 It has a parameter, a CK_SP800_108_KDF_PARAMS structure.

11832 The following table lists the data field types that are supported for this KDF type and their meaning:

11833 Table 197, 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>

	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.

11834

11835 SP800-108 limits the amount of derived keying material that can be produced by a Double-Pipeline Mode
11836 KDF by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be
11837 produced is $(2^{32}-1)h$, where "h" is the length in bits of the output of the selected PRF.

11838 The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method
11839 used to construct the PRF input data; the data/values identified by the array of **CK_PRF_DATA_PARAM**
11840 structures are concatenated in to a byte array that is used as the IV. As shown in SP800-108 section 5.3,
11841 the CK_SP800_108_ITERATION_VARIABLE and CK_SP800_108_COUNTER data field types are not
11842 included in IV construction process. All other data field types are included in the construction process.

11843 6.42.6 Deriving Additional Keys

11844 The KDFs defined in this section can be used to derive more than one symmetric key from the base key.
11845 The **C_Derive** function accepts one CK_ATTRIBUTE_PTR to define a single derived key and one
11846 CK_OBJECT_HANDLE_PTR to receive the handle for the derived key.

11847 To derive additional keys, the mechanism parameter structure can be filled in with one or more
11848 CK_DERIVED_KEY structures. Each structure contains a CK_ATTRIBUTE_PTR to define a derived key
11849 and a CK_OBJECT_HANDLE_PTR to receive the handle for the additional derived keys. The key
11850 defined by the **C_Derive** function parameters is always derived before the keys defined by the
11851 CK_DERIVED_KEY array that is part of the mechanism parameter. The additional keys that are defined
11852 by the CK_DERIVED_KEY array are derived in the order they are defined in the array. That is to say that
11853 the derived keying material produced by the KDF is processed from left to right, and bytes are assigned
11854 first to the key defined by the **C_Derive** function parameters, and then bytes are assigned to the keys that
11855 are defined by the CK_DERIVED_KEY array in the order they are defined in the array.

11856 Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration
11857 will produce enough keying material for the key being derived. Other times, additional internal iterations
11858 are performed to produce multiple segments which are concatenated together to produce enough keying
11859 material for the derived key(s).

11860 When deriving multiple keys, no key can be created using part of a segment that was used for another
11861 key. All keys must be created from disjoint segments. For example, if the parameters are defined such

that a 48-byte key (defined by the **C_Derive** function parameters) and a 16-byte key (defined by the content of CK_DERIVED_KEY) are to be derived using **CKM_SHA256_HMAC** as a PRF, three internal iterations of the KDF will be performed and three segments of PRF output will be produced. The first segment and half of the second segment will be used to create the 48-byte key and the third segment will be used to create the 16-byte key.

3 KDF Segments of Output:

32-byte segment	32-byte segment	32-byte segment
-----------------	-----------------	-----------------

2 Derived Keys:

48-byte key	unused	16-byte key	unused
-------------	--------	-------------	--------

In the above example, if the CK_SP800_108_DKM_LENGTH data field type is specified with method CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, then the DKM length value will be 512 bits. If the CK_SP800_108_DKM_LENGTH data field type is specified with method CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS, then the DKM length value will be 768 bits.

When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall be derived. If the failure was caused by the content of a specific key's template (ie the template defined by the content of *pTemplate*), the corresponding *phKey* value will be set to CK_INVALID_HANDLE to identify the offending template.

6.42.7 Key Derivation Attribute Rules

The **CKM_SP800_108_COUNTER_KDF**, **CKM_SP800_108_FEEDBACK_KDF** and **CKM_SP800_108_DOUBLE_PIPELINE_KDF** mechanisms have the following rules about key sensitivity and extractability:

- The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key(s) 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.

6.42.8 Constructing PRF Input Data

SP800-108 defines the PRF input data for each KDF at a high level using terms like “label”, “context”, “separator”, “counter”...etc. The value, formatting and order of the input data is not strictly defined by SP800-108, instead it is described as being defined by the “encoding scheme”.

To support any encoding scheme, these mechanisms construct the PRF input data from the array of CK_PRF_DATA_PARAM structures in the mechanism parameter. All of the values defined by the CK_PRF_DATA_PARAM array are concatenated in the order they are defined and passed in to the PRF as the data parameter.

6.42.8.1 Sample Counter Mode KDF

SP800-108 section 5.1 outlines a sample Counter Mode KDF which defines the following PRF input:

PRF (K_I , $[i]_2 || Label || 0x00 || Context || [L]_2$)

Section 5.1 does not define the number of bits used to represent the counter (the “r” value) or the DKM length (the “L” value), so 16-bits is assumed for both cases. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```

11905     #define DIM(a) (sizeof((a))/sizeof((a)[0]))
11906
11907     CK_OBJECT_HANDLE hBaseKey;
11908     CK_OBJECT_HANDLE hDerivedKey;
11909     CK_ATTRIBUTE derivedKeyTemplate = { ... };
11910
11911     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
11912     CK_ULONG ulLabelLen = sizeof(baLabel);
11913     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
11914     CK_ULONG ulContextLen = sizeof(baContext);
11915
11916     CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
11917     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
11918         = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
11919
11920     CK_PRF_DATA_PARAM dataParams[] =
11921     {
11922         { CK_SP800_108_ITERATION_VARIABLE,
11923           &counterFormat, sizeof(counterFormat) },
11924         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
11925         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
11926         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
11927         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
11928     };
11929
11930     CK_SP800_108_KDF_PARAMS kdfParams =
11931     {
11932         CKM_AES_CMAC,
11933         DIM(dataParams),
11934         &dataParams,
11935         0,          /* no addition derived keys */
11936         NULL        /* no addition derived keys */
11937     };
11938
11939     CK_MECHANISM = mechanism
11940     {
11941         CKM_SP800_108_COUNTER_KDF,
11942         &kdfParams,
11943         sizeof(kdfParams)
11944     };
11945
11946     hBaseKey = GetBaseKeyHandle(.....);
11947
11948     rv = C_DeriveKey(
11949         hSession,
11950         &mechanism,
11951         hBaseKey,
11952         &derivedKeyTemplate,
11953         DIM(derivedKeyTemplate),
11954         &hDerivedKey);

```

11955 6.42.8.2 Sample SCP03 Counter Mode KDF

11956 The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:

11957 $PRF(K_I, Label || 0x00 || [L]_2 || [i]_2 || Context)$

SCP03 defines the number of bits used to represent the counter (the “r” value) and number of bits used to represent the DKM length (the “L” value) as 16-bits. The following sample code shows how to define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```
#define DIM(a) (sizeof((a))/sizeof((a)[0]))

CK_OBJECT_HANDLE hBaseKey;
CK_OBJECT_HANDLE hDerivedKey;
CK_ATTRIBUTE derivedKeyTemplate = { ... };

CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
CK_ULONG ulLabelLen = sizeof(baLabel);
CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
CK_ULONG ulContextLen = sizeof(baContext);

CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
    = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};

CK_PRF_DATA_PARAM dataParams[] =
{
    { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
    { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
    { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },
    { CK_SP800_108_ITERATION_VARIABLE,
      &counterFormat, sizeof(counterFormat) },
    { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }
};

CK_SP800_108_KDF_PARAMS kdfParams =
{
    CKM_AES_CMAC,
    DIM(dataParams),
    &dataParams,
    0,          /* no addition derived keys */
    NULL       /* no addition derived keys */
};

CK_MECHANISM = mechanism
{
    CKM_SP800_108_COUNTER_KDF,
    &kdfParams,
    sizeof(kdfParams)
};

hBaseKey = GetBaseKeyHandle(.....);

rv = C_DeriveKey(
    hSession,
    &mechanism,
    hBaseKey,
    &derivedKeyTemplate,
    DIM(derivedKeyTemplate),
    &hDerivedKey);
```

6.42.8.3 Sample Feedback Mode KDF

SP800-108 section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF input:

12013 PRF (K_i , $K(i-1)$ $\{ || [i]_2 \} ||$ *Label* $|| 0x00 ||$ *Context* $|| [L]_2$)

12014 Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM
12015 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and
12016 is included in this example. The following sample code shows how to define this PRF input data using an
12017 array of CK_PRF_DATA_PARAM structures.

```
12018       #define DIM(a) (sizeof((a))/sizeof((a)[0]))
12019
12020       CK_OBJECT_HANDLE hBaseKey;
12021       CK_OBJECT_HANDLE hDerivedKey;
12022       CK_ATTRIBUTE derivedKeyTemplate = { ... };
12023
12024       CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};
12025       CK_ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);
12026       CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
12027       CK_ULONG ulLabelLen = sizeof(baLabel);
12028       CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
12029       CK_ULONG ulContextLen = sizeof(baContext);
12030
12031       CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
12032       CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
12033       = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
12034
12035       CK_PRF_DATA_PARAM dataParams[] =
12036       {
12037           { CK_SP800_108_ITERATION_VARIABLE,
12038             &counterFormat, sizeof(counterFormat) },
12039           { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
12040           { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
12041           { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
12042           { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
12043       };
12044
12045       CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams =
12046       {
12047           CKM_AES_CMAC,
12048           DIM(dataParams),
12049           &dataParams,
12050           ulFeedbackIVLen,
12051           baFeedbackIV,
12052           0,       /* no addition derived keys */
12053           NULL     /* no addition derived keys */
12054       };
12055
12056       CK_MECHANISM = mechanism
12057       {
12058           CKM_SP800_108_FEEDBACK_KDF,
12059           &kdfParams,
12060           sizeof(kdfParams)
12061       };
12062
12063       hBaseKey = GetBaseKeyHandle(.....);
12064
12065       rv = C_DeriveKey(
12066           hSession,
12067           &mechanism,
12068           hBaseKey,
```

```

12069         &derivedKeyTemplate,
12070         DIM(derivedKeyTemplate),
12071         &hDerivedKey);

```

12072 6.42.8.4 Sample Double-Pipeline Mode KDF

12073 SP800-108 section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two following
12074 PRF inputs:

```

12075         PRF (KI, A(i-1))
12076         PRF (KI, K(i-1) || [i]2 || Label || 0x00 || Context || [L]2)

```

12077 Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM
12078 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it
12079 is left out in this example. The following sample code shows how to define this PRF input data using an
12080 array of CK_PRF_DATA_PARAM structures.

```

12081     #define DIM(a) (sizeof((a))/sizeof((a)[0]))
12082
12083     CK_OBJECT_HANDLE hBaseKey;
12084     CK_OBJECT_HANDLE hDerivedKey;
12085     CK_ATTRIBUTE derivedKeyTemplate = { ... };
12086
12087     CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
12088     CK_ULONG ulLabelLen = sizeof(baLabel);
12089     CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
12090     CK_ULONG ulContextLen = sizeof(baContext);
12091
12092     CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
12093     = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
12094
12095     CK_PRF_DATA_PARAM dataParams[] =
12096     {
12097         { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
12098         { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
12099         { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
12100         { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
12101     };
12102
12103     CK_SP800_108_KDF_PARAMS kdfParams =
12104     {
12105         CKM_AES_CMAC,
12106         DIM(dataParams),
12107         &dataParams,
12108         0, /* no addition derived keys */
12109         NULL /* no addition derived keys */
12110     };
12111
12112     CK_MECHANISM = mechanism
12113     {
12114         CKM_SP800_108_DOUBLE_PIPELINE_KDF,
12115         &kdfParams,
12116         sizeof(kdfParams)
12117     };
12118
12119     hBaseKey = GetBaseKeyHandle(.....);
12120
12121     rv = C_DeriveKey(
12122         hSession,

```

```
12123     &mechanism,  
12124     hBaseKey,  
12125     &derivedKeyTemplate,  
12126     DIM(derivedKeyTemplate),  
12127     &hDerivedKey);
```

12128 **6.43 Miscellaneous simple key derivation mechanisms**

12129 *Table 198, 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							✓

12130 **6.43.1 Definitions**

- 12131 Mechanisms:
- 12132 CKM_CONCATENATE_BASE_AND_DATA
 - 12133 CKM_CONCATENATE_DATA_AND_BASE
 - 12134 CKM_XOR_BASE_AND_DATA
 - 12135 CKM_EXTRACT_KEY_FROM_KEY
 - 12136 CKM_CONCATENATE_BASE_AND_KEY

12137 **6.43.2 Parameters for miscellaneous simple key derivation mechanisms**

12138 ♦ **CK_KEY_DERIVATION_STRING_DATA;**
12139 **CK_KEY_DERIVATION_STRING_DATA_PTR**

12140 CK_KEY_DERIVATION_STRING_DATA provides the parameters for the
12141 CKM_CONCATENATE_BASE_AND_DATA, CKM_CONCATENATE_DATA_AND_BASE, and
12142 CKM_XOR_BASE_AND_DATA mechanisms. It is defined as follows:

```
12143     typedef struct CK_KEY_DERIVATION_STRING_DATA {  
12144         CK_BYTE_PTR pData;  
12145         CK_ULONG ulLen;  
12146     } CK_KEY_DERIVATION_STRING_DATA;
```

12147
12148 The fields of the structure have the following meanings:

- 12149 pData pointer to the byte string
- 12150 ulLen length of the byte string

12151 **CK_KEY_DERIVATION_STRING_DATA_PTR** is a pointer to a
12152 **CK_KEY_DERIVATION_STRING_DATA**.

12153 ♦ **CK_EXTRACT_PARAMS; CK_EXTRACT_PARAMS_PTR**

12154 **CK_EXTRACT_PARAMS** provides the parameter to the **CKM_EXTRACT_KEY_FROM_KEY**
12155 mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is
12156 defined as follows:

```
12157     typedef CK_ULONG CK_EXTRACT_PARAMS;
```

12158

12159 **CK_EXTRACT_PARAMS_PTR** is a pointer to a **CK_EXTRACT_PARAMS**.

12160 **6.43.3 Concatenation of a base key and another key**

12161 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_KEY**, derives a secret key from the
12162 concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys
12163 specified are concatenated together in a buffer.

12164 This mechanism takes a parameter, a **CK_OBJECT_HANDLE**. This handle produces the key value
12165 information which is appended to the end of the base key's value information (the base key is the key
12166 whose handle is supplied as an argument to **C_DeriveKey**).

12167 For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF,
12168 then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 12169 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
12170 generic secret key. Its length will be equal to the sum of the lengths of the values of the two original
12171 keys.
- 12172 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
12173 will be a generic secret key of the specified length.
- 12174 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
12175 length. If it does, then the key produced by this mechanism will be of the type specified in the
12176 template. If it doesn't, an error will be returned.
- 12177 • If both a key type and a length are provided in the template, the length must be compatible with that
12178 key type. The key produced by this mechanism will be of the specified type and length.

12179 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
12180 properly.

12181 If the requested type of key requires more bytes than are available by concatenating the two original keys'
12182 values, an error is generated.

12183 This mechanism has the following rules about key sensitivity and extractability:

- 12184 • If either of the two original keys has its **CKA_SENSITIVE** attribute set to **CK_TRUE**, so does the
12185 derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied
12186 template or from a default value.
- 12187 • Similarly, if either of the two original keys has its **CKA_EXTRACTABLE** attribute set to **CK_FALSE**,
12188 so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either
12189 from the supplied template or from a default value.
- 12190 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to **CK_TRUE** if and only if both of the
12191 original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to **CK_TRUE**.
- 12192 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to **CK_TRUE** if and only if
12193 both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to **CK_TRUE**.

12194 **6.43.4 Concatenation of a base key and data**

12195 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by
12196 concatenating data onto the end of a specified secret key.

12197 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
 12198 specifies the length and value of the data which will be appended to the base key to derive another key.
 12199 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
 12200 the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 12201 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
 12202 generic secret key. Its length will be equal to the sum of the lengths of the value of the original key
 12203 and the data.
- 12204 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
 12205 will be a generic secret key of the specified length.
- 12206 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
 12207 length. If it does, then the key produced by this mechanism will be of the type specified in the
 12208 template. If it doesn't, an error will be returned.
- 12209 • If both a key type and a length are provided in the template, the length must be compatible with that
 12210 key type. The key produced by this mechanism will be of the specified type and length.

12211 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
 12212 properly.

12213 If the requested type of key requires more bytes than are available by concatenating the original key's
 12214 value and the data, an error is generated.

12215 This mechanism has the following rules about key sensitivity and extractability:

- 12216 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
 12217 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
 12218 default value.
- 12219 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 12220 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
 12221 supplied template or from a default value.
- 12222 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
 12223 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 12224 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 12225 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

12226 6.43.5 Concatenation of data and a base key

12227 This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by
 12228 prepending data to the start of a specified secret key.

12229 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
 12230 specifies the length and value of the data which will be prepended to the base key to derive another key.

12231 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
 12232 the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.

- 12233 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
 12234 generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the
 12235 original key.
- 12236 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
 12237 will be a generic secret key of the specified length.
- 12238 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
 12239 length. If it does, then the key produced by this mechanism will be of the type specified in the
 12240 template. If it doesn't, an error will be returned.
- 12241 • If both a key type and a length are provided in the template, the length must be compatible with that
 12242 key type. The key produced by this mechanism will be of the specified type and length.

12243 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
 12244 properly.

12245 If the requested type of key requires more bytes than are available by concatenating the data and the
 12246 original key's value, an error is generated.

12247 This mechanism has the following rules about key sensitivity and extractability:

- 12248 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
 12249 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
 12250 default value.
- 12251 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 12252 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
 12253 supplied template or from a default value.
- 12254 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
 12255 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 12256 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 12257 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

12258 6.43.6 XORing of a key and data

12259 XORing key derivation, denoted **CKM_XOR_BASE_AND_DATA**, is a mechanism which provides the
 12260 capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle
 12261 and some data.

12262 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
 12263 specifies the data with which to XOR the original key's value.

12264 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
 12265 the value of the derived key will be taken from a buffer containing the string 0x88888888.

- 12266 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
 12267 generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of
 12268 the original key.
- 12269 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
 12270 will be a generic secret key of the specified length.
- 12271 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
 12272 length. If it does, then the key produced by this mechanism will be of the type specified in the
 12273 template. If it doesn't, an error will be returned.
- 12274 • If both a key type and a length are provided in the template, the length must be compatible with that
 12275 key type. The key produced by this mechanism will be of the specified type and length.

12276 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
 12277 properly.

12278 If the requested type of key requires more bytes than are available by taking the shorter of the data and
 12279 the original key's value, an error is generated.

12280 This mechanism has the following rules about key sensitivity and extractability:

- 12281 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
 12282 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
 12283 default value.
- 12284 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 12285 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
 12286 supplied template or from a default value.
- 12287 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
 12288 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 12289 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 12290 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

6.43.7 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**.

6.44 CMS

Table 199, CMS Mechanisms vs. Functions

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

6.44.1 Definitions

Mechanisms:

CKM_CMS_SIG

6.44.2 CMS Signature Mechanism Objects

These objects provide information relating to the CKM_CMS_SIG mechanism. CKM_CMS_SIG mechanism object attributes represent information about supported CMS signature attributes in the token. They are only present on tokens supporting the **CKM_CMS_SIG** mechanism, but must be present on those tokens.

Table 200, 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

The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-encoded. The list of attributes is defined in ASN.1 as:

```

Attributes ::= SET SIZE (1..MAX) OF Attribute
Attribute ::= SEQUENCE {
    attrType      OBJECT IDENTIFIER,
    attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER
                OPTIONAL
}

```

The client may not set any of the attributes.

6.44.3 CMS mechanism parameters

• CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR

CK_CMS_SIG_PARAMS is a structure that provides the parameters to the **CKM_CMS_SIG** mechanism. It is defined as follows:

```

typedef struct CK_CMS_SIG_PARAMS {
    CK_OBJECT_HANDLE      certificateHandle;
    CK_MECHANISM_PTR      pSigningMechanism;
    CK_MECHANISM_PTR      pDigestMechanism;
}

```

```

12359     CK_UTF8CHAR_PTR      pContentType;
12360     CK_BYTE_PTR           pRequestedAttributes;
12361     CK_ULONG              ulRequestedAttributesLen;
12362     CK_BYTE_PTR           pRequiredAttributes;
12363     CK_ULONG              ulRequiredAttributesLen;
12364 } CK_CMS_SIG_PARAMS;

```

12365

12366 The fields of the structure have the following meanings:

12367 12368 12369 12370 12371 12372	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.
12373 12374	pSigningMechanism	Mechanism to use when signing a constructed CMS SignedAttributes value. E.g. CKM_SHA1_RSA_PKCS .
12375 12376 12377	pDigestMechanism	Mechanism to use when digesting the data. Value shall be NULL_PTR when the digest mechanism to use follows from the pSigningMechanism parameter.
12378 12379 12380 12381 12382 12383 12384 12385 12386 12387	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.
12388 12389 12390	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.
12391	ulRequestedAttributesLen	Length in bytes of the value pointed to by pRequestedAttributes
12392 12393 12394 12395 12396 12397 12398	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.
12399	ulRequiredAttributesLen	Length in bytes, of the value pointed to by pRequiredAttributes.

12400 6.44.4 CMS signatures

12401 The CMS mechanism, denoted **CKM_CMS_SIG**, is a multi-purpose mechanism based on the structures
12402 defined in PKCS #7 and RFC 2630. It supports single- or multiple-part signatures with and without
12403 message recovery. The mechanism is intended for use with, e.g., PTDs (see MeT-PTD) or other capable
12404 tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this value.
12405 The content of the **SignedAttributes** value is decided by the token, however the caller can suggest some
12406 attributes in the parameter *pRequestedAttributes*. The caller can also require some attributes to be

present through the parameters *pRequiredAttributes*. The signature is computed in accordance with the parameter *pSigningMechanism*.

When this mechanism is used in successful calls to **C_Sign** or **C_SignFinal**, the *pSignature* return value will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a complete definition of all fields and types, see RFC 2630):

```
SignerInfo ::= SEQUENCE {  
    version CMSVersion,  
    sid SignerIdentifier,  
    digestAlgorithm DigestAlgorithmIdentifier,  
    signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,  
    signatureAlgorithm SignatureAlgorithmIdentifier,  
    signature SignatureValue,  
    unsignedAttrs [1] IMPLICIT UnsignedAttributes  
    OPTIONAL }
```

The *certificateHandle* parameter, when set, helps the token populate the **sid** field of the **SignerInfo** value. If *certificateHandle* is **NULL_PTR** the choice of a suitable certificate reference in the **SignerInfo** result value is left to the token (the token could, e.g., interact with the user).

This mechanism shall not be used in calls to **C_Verify** or **C_VerifyFinal** (use the *pSigningMechanism* mechanism instead).

For the *pRequiredAttributes* field, the token may have to interact with the user to find out whether to accept a proposed value or not. The token should never accept any proposed attribute values without some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings and not direct interaction). If a user rejects proposed values, or the signature request as such, the value **CKR_FUNCTION_REJECTED** shall be returned.

When possible, applications should use the **CKM_CMS_SIG** mechanism when generating CMS-compatible signatures rather than lower-level mechanisms such as **CKM_SHA1_RSA_PKCS**. This is especially true when the signatures are to be made on content that the token is able to present to a user. Exceptions may include those cases where the token does not support a particular signing attribute. Note however that the token may refuse usage of a particular signature key unless the content to be signed is known (i.e. the **CKM_CMS_SIG** mechanism is used).

When a token does not have presentation capabilities, the PKCS #11-aware application may avoid sending the whole message to the token by electing to use a suitable signature mechanism (e.g. **CKM_RSA_PKCS**) as the *pSigningMechanism* value in the **CK_CMS_SIG_PARAMS** structure, and digesting the message itself before passing it to the token.

PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to provide messages to be signed by the token in a format possible for the token to present to the user. Tokens that receive multipart MIME-messages for which only certain parts are possible to present may fail the signature operation with a return value of **CKR_DATA_INVALID**, but may also choose to add a signing attribute indicating which parts of the message were possible to present.

6.45 Blowfish

Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times. The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex initialization phase required before any encryption can take place, the actual encryption of data is very efficient on large microprocessors.

Table 201, 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	✓					✓	

6.45.1 Definitions

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

Mechanisms:

CKM_BLOWFISH_KEY_GEN

CKM_BLOWFISH_CBC

CKM_BLOWFISH_CBC_PAD

6.45.2 BLOWFISH secret key objects

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

Table 202, 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

Refer to Table 11 for footnotes

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

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_BLOWFISH;
CK_UTF8CHAR label[] = "A blowfish secret key object";
CK_BYTE value[16] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

```

6.45.3 Blowfish key generation

The Blowfish key generation mechanism, denoted **CKM_BLOWFISH_KEY_GEN**, is a key generation mechanism Blowfish.

It does not have a parameter.

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

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the 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.

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

6.45.4 Blowfish-CBC

Blowfish-CBC, denoted **CKM_BLOWFISH_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping.

It has a parameter, a 8-byte initialization vector.

This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus one null bytes so that the resulting length is a multiple of the block size. The output data is the same length as the padded input data. It does not wrap the key type, key length, or any other information about the key; the application must convey these separately.

For unwrapping, the mechanism decrypts the wrapped key, 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 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.

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

Table 203, 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

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

6.45.5 Blowfish-CBC with PKCS padding

Blowfish-CBC-PAD, denoted **CKM_BLOWFISH_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block cipher padding method detailed in PKCS #7.

It has a parameter, a 8-byte initialization vector.

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

The entries in the table below for data length constraints when wrapping and unwrapping keys do not apply to wrapping and unwrapping private keys.

12518 Constraints on key types and the length of data are summarized in the following table:

12519

12520 *Table 204, 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

12521 6.46 Twofish

12522 Ref. <https://www.schneier.com/twofish.html>

12523 6.46.1 Definitions

12524 This section defines the key type “CKK_TWOFISH” for type CK_KEY_TYPE as used in the
12525 CKA_KEY_TYPE attribute of key objects.

12526 Mechanisms:

12527 CKM_TWOFISH_KEY_GEN

12528 CKM_TWOFISH_CBC

12529 CKM_TWOFISH_CBC_PAD

12530

12531 6.46.2 Twofish secret key objects

12532 Twofish secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_TWOFISH**) hold Twofish
12533 keys. The following table defines the Twofish secret key object attributes, in addition to the common
12534 attributes defined for this object class:

12535 *Table 205, 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

12536 Refer to Table 11 for footnotes

12537 The following is a sample template for creating an TWOFISH secret key object:

```
12538 CK_OBJECT_CLASS class = CKO_SECRET_KEY;  
12539 CK_KEY_TYPE keyType = CKK_TWOFISH;  
12540 CK_UTF8CHAR label[] = "A twofish secret key object";  
12541 CK_BYTE value[16] = {...};  
12542 CK_BBOOL true = CK_TRUE;  
12543 CK_ATTRIBUTE template[] = {  
12544     {CKA_CLASS, &class, sizeof(class)},
```



```

12545     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12546     {CKA_TOKEN, &true, sizeof(true)},
12547     {CKA_LABEL, label, sizeof(label)-1},
12548     {CKA_ENCRYPT, &true, sizeof(true)},
12549     {CKA_VALUE, value, sizeof(value)}
12550 };

```

12551 6.46.3 Twofish key generation

12552 The Twofish key generation mechanism, denoted **CKM_TWOFISH_KEY_GEN**, is a key generation
 12553 mechanism Twofish.

12554 It does not have a parameter.

12555 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN**
 12556 attribute of the template for the key.

12557 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 12558 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
 12559 supports) may be specified in the template for the key, or else are assigned default initial values.

12560 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12561 specify the supported range of key sizes, in bytes.

12562 6.46.4 Twofish -CBC

12563 Twofish-CBC, denoted **CKM_TWOFISH_CBC**, is a mechanism for single- and multiple-part encryption
 12564 and decryption; key wrapping; and key unwrapping.

12565 It has a parameter, a 16-byte initialization vector.

12566 6.46.5 Twofish-CBC with PKCS padding

12567 Twofish-CBC-PAD, denoted **CKM_TWOFISH_CBC_PAD**, is a mechanism for single- and multiple-part
 12568 encryption and decryption, key wrapping and key unwrapping, cipher-block chaining mode and the block
 12569 cipher padding method detailed in PKCS #7.

12570 It has a parameter, a 16-byte initialization vector.

12571 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
 12572 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
 12573 the **CKA_VALUE_LEN** attribute.

12574 6.47 CAMELLIA

12575 Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.
 12576 Camellia is described e.g. in IETF RFC 3713.

12577 *Table 206, Camellia Mechanisms vs. Functions*

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen · Key / Key Pair	Wrap & Unwra p	Deriv e
CKM_CAMELLIA_KEY_GEN					✓		
CKM_CAMELLIA_ECB	✓					✓	

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen · Key / Key Pair	Wrap & Unwra p	Deriv e
CKM_CAMELLIA_CBC	✓					✓	
CKM_CAMELLIA_CBC_PAD	✓					✓	
CKM_CAMELLIA_MAC_GENERAL		✓					
CKM_CAMELLIA_MAC		✓					
CKM_CAMELLIA_ECB_ENCRYPT_DATA							✓
CKM_CAMELLIA_CBC_ENCRYPT_DATA							✓

6.47.1 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

6.47.2 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 207, 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 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[] = {
```

```

12601     {CKA_CLASS, &class, sizeof(class)},
12602     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12603     {CKA_TOKEN, &true, sizeof(true)},
12604     {CKA_LABEL, label, sizeof(label)-1},
12605     {CKA_ENCRYPT, &true, sizeof(true)},
12606     {CKA_VALUE, value, sizeof(value)}
12607 };

```

12608 6.47.3 Camellia key generation

12609 The Camellia key generation mechanism, denoted CKM_CAMELLIA_KEY_GEN, is a key generation
 12610 mechanism for Camellia.

12611 It does not have a parameter.

12612 The mechanism generates Camellia keys with a particular length in bytes, as specified in the
 12613 **CKA_VALUE_LEN** attribute of the template for the key.

12614 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
 12615 key. Other attributes supported by the Camellia key type (specifically, the flags indicating which functions
 12616 the key supports) may be specified in the template for the key, or else are assigned default initial values.

12617 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 12618 specify the supported range of Camellia key sizes, in bytes.

12619 6.47.4 Camellia-ECB

12620 Camellia-ECB, denoted **CKM_CAMELLIA_ECB**, is a mechanism for single- and multiple-part encryption
 12621 and decryption; key wrapping; and key unwrapping, based on Camellia and electronic codebook mode.

12622 It does not have a parameter.

12623 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
 12624 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
 12625 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
 12626 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
 12627 length as the padded input data. It does not wrap the key type, key length, or any other information about
 12628 the key; the application must convey these separately.

12629 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
 12630 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
 12631 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
 12632 attribute of the new key; other attributes required by the key type must be specified in the template.

12633 Constraints on key types and the length of data are summarized in the following table:

12634 *Table 208, 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	

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

6.47.5 Camellia-CBC

Camellia-CBC, denoted **CKM_CAMELLIA_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.

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

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

For unwrapping, the mechanism decrypts the wrapped key, 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 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.

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

Table 209, 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	

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

6.47.6 Camellia-CBC with PKCS padding

Camellia-CBC with PKCS padding, denoted **CKM_CAMELLIA_CBC_PAD**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia; cipher-block chaining mode; and the block cipher padding method detailed in PKCS #7.

12659 It has a parameter, a 16-byte initialization vector.

12660 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the

12661 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified

12662 for the **CKA_VALUE_LEN** attribute.

12663 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,

12664 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for

12665 details). The entries in the table below for data length constraints when wrapping and unwrapping keys

12666 do not apply to wrapping and unwrapping private keys.

12667 Constraints on key types and the length of data are summarized in the following table:

12668 *Table 210, 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 size 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

12669 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure

12670 specify the supported range of Camellia key sizes, in bytes.

12671

12672 6.47.7 CAMELLIA with Counter mechanism parameters

12673 ♦ **CK_CAMELLIA_CTR_PARAMS; CK_CAMELLIA_CTR_PARAMS_PTR**

12674 **CK_CAMELLIA_CTR_PARAMS** is a structure that provides the parameters to the

12675 **CKM_CAMELLIA_CTR** mechanism. It is defined as follows:

```
12676     typedef struct CK_CAMELLIA_CTR_PARAMS {
12677         CK_ULONG ulCounterBits;
12678         CK_BYTE cb[16];
12679     } CK_CAMELLIA_CTR_PARAMS;
```

12681 *ulCounterBits* specifies the number of bits in the counter block (cb) that shall be incremented. This

12682 number shall be such that $0 < ulCounterBits \leq 128$. For any values outside this range the mechanism

12683 shall return **CKR_MECHANISM_PARAM_INVALID**.

12684 It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter

12685 bits are the least significant bits of the counter block (cb). They are a big-endian value usually starting

12686 with 1. The rest of 'cb' is for the nonce, and maybe an optional IV.

12687 E.g. as defined in [RFC 3686]:

```

12688      0              1              2              3
12689      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
12690      +-----+-----+-----+-----+-----+-----+-----+-----+
12691      |                                     Nonce                                     |
12692      +-----+-----+-----+-----+-----+-----+-----+-----+
12693      |                               Initialization Vector (IV)                               |
12694      |                                                                                               |
12695      +-----+-----+-----+-----+-----+-----+-----+-----+
12696      |                                     Block Counter                                     |
12697      +-----+-----+-----+-----+-----+-----+-----+-----+
12698

```

12699	This construction permits each packet to consist of up to $2^{32}-1$ blocks = 4,294,967,295 blocks =
12700	68,719,476,720 octets.

12701 **CK_CAMELLIA_CTR_PARAMS_PTR** is a pointer to a **CK_CAMELLIA_CTR_PARAMS**.

12702

12703 6.47.8 General-length Camellia-MAC

12704 General-length Camellia -MAC, denoted CKM_CAMELLIA_MAC_GENERAL, is a mechanism for single-
12705 and multiple-part signatures and verification, based on Camellia and data authentication as defined
12706 in.[CAMELLIA]

12705 and multiple-part signatures and verification, based on Camellia and data authentication as defined
12706 in.[CAMELLIA]

12707 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
12708 desired from the mechanism.

12708 desired from the mechanism.

12709 The output bytes from this mechanism are taken from the start of the final Camellia cipher block produced
12710 in the MACing process.

12710 in the MACing process.

12711 Constraints on key types and the length of data are summarized in the following table:

12712 *Table 211, 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

12713 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12714 specify the supported range of Camellia key sizes, in bytes.

12714 specify the supported range of Camellia key sizes, in bytes.

12715 **6.47.9 Camellia-MAC**

12716 Camellia-MAC, denoted by **CKM_CAMELLIA_MAC**, is a special case of the general-length Camellia-
12717 MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in
12718 length.

12717 MAC mechanism. Camellia-MAC always produces and verifies MACs that are half the block size in
12718 length.

12719 It does not have a parameter.

12720 Constraints on key types and the length of data are summarized in the following table:

12721 *Table 212, Camellia-MAC: Key and Data Length*

Function	Key type	Data length	Signature length
C_Sign	CKK_CAMELLIA	any	½ block size (8 bytes)
C_Verify	CKK_CAMELLIA	any	½ block size (8 bytes)

12722 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12723 specify the supported range of Camellia key sizes, in bytes.

12723 specify the supported range of Camellia key sizes, in bytes.

6.48 Key derivation by data encryption - Camellia

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

6.48.1 Definitions

Mechanisms:

CKM_CAMELLIA_ECB_ENCRYPT_DATA

CKM_CAMELLIA_CBC_ENCRYPT_DATA

```
typedef struct CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS {  
    CK_BYTE      iv[16];  
    CK_BYTE_PTR  pData;  
    CK_ULONG     length;  
} CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS;
```

```
typedef CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.48.2 Mechanism Parameters

Uses CK_CAMELLIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

Table 213, 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.

6.49 ARIA

ARIA is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES. ARIA is described in NSRI "Specification of ARIA".

Table 214, 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							✓

6.49.1 Definitions

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

Mechanisms:

CKM_ARIA_KEY_GEN
CKM_ARIA_ECB
CKM_ARIA_CBC
CKM_ARIA_MAC
CKM_ARIA_MAC_GENERAL
CKM_ARIA_CBC_PAD

6.49.2 Aria secret key objects

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

Table 215, 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

Refer to Table 11 for footnotes

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

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_ARIA;
CK_UTF8CHAR label[] = "An ARIA secret key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
}
```

12776 {CKA_VALUE, value, sizeof(value)}
12777 };

12778 6.49.3 ARIA key generation

12779 The ARIA key generation mechanism, denoted CKM_ARIA_KEY_GEN, is a key generation mechanism
12780 for Aria.

12781 It does not have a parameter.

12782 The mechanism generates ARIA keys with a particular length in bytes, as specified in the
12783 **CKA_VALUE_LEN** attribute of the template for the key.

12784 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12785 key. Other attributes supported by the ARIA key type (specifically, the flags indicating which functions the
12786 key supports) may be specified in the template for the key, or else are assigned default initial values.

12787 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12788 specify the supported range of ARIA key sizes, in bytes.

12789 6.49.4 ARIA-ECB

12790 ARIA-ECB, denoted **CKM_ARIA_ECB**, is a mechanism for single- and multiple-part encryption and
12791 decryption; key wrapping; and key unwrapping, based on Aria and electronic codebook mode.

12792 It does not have a parameter.

12793 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
12794 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
12795 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
12796 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
12797 length as the padded input data. It does not wrap the key type, key length, or any other information about
12798 the key; the application must convey these separately.

12799 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
12800 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
12801 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
12802 attribute of the new key; other attributes required by the key type must be specified in the template.

12803 Constraints on key types and the length of data are summarized in the following table:

12804 Table 216, 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	

12805 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12806 specify the supported range of ARIA key sizes, in bytes.

12807 6.49.5 ARIA-CBC

12808 ARIA-CBC, denoted **CKM_ARIA_CBC**, is a mechanism for single- and multiple-part encryption and
12809 decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

12810 It has a parameter, a 16-byte initialization vector.

12811 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to

12812 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the

12813 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus

12814 one null bytes so that the resulting length is a multiple of the block size. The output data is the same

12815 length as the padded input data. It does not wrap the key type, key length, or any other information about

12816 the key; the application must convey these separately.

12817 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the

12818 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the

12819 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**

12820 attribute of the new key; other attributes required by the key type must be specified in the template.

12821 Constraints on key types and the length of data are summarized in the following table:

12822 *Table 217, 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	

12823 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure

12824 specify the supported range of Aria key sizes, in bytes.

12825 6.49.6 ARIA-CBC with PKCS padding

12826 ARIA-CBC with PKCS padding, denoted **CKM_ARIA_CBC_PAD**, is a mechanism for single- and

12827 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block

12828 chaining mode; and the block cipher padding method detailed in PKCS #7.

12829 It has a parameter, a 16-byte initialization vector.

12830 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the

12831 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified

12832 for the **CKA_VALUE_LEN** attribute.

12833 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,

12834 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see Section 6.7 for

12835 details). The entries in the table below for data length constraints when wrapping and unwrapping keys

12836 do not apply to wrapping and unwrapping private keys.

12837 Constraints on key types and the length of data are summarized in the following table:

12838 *Table 218, 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

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

6.49.7 General-length ARIA-MAC

General-length ARIA -MAC, denoted **CKM_ARIA_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

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

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

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

Table 219, 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

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

6.49.8 ARIA-MAC

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

It does not have a parameter.

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

Table 220, 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)

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

6.50 Key derivation by data encryption - ARIA

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

6.50.1 Definitions

Mechanisms:

CKM_ARIA_ECB_ENCRYPT_DATA

CKM_ARIA_CBC_ENCRYPT_DATA

```
typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {
    CK_BYTE      iv[16];
    CK_BYTE_PTR  pData;
    CK_ULONG     length;
} CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;
```

```
typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR
      CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.50.2 Mechanism Parameters

Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

Table 221, 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.

6.51 SEED

SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It has a 128-bit key size and a 128-bit block size.

Its specification has been published as Internet [RFC 4269].

RFCs have been published defining the use of SEED in

TLS <ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt>

IPsec <ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt>

CMS <ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt>

TLS cipher suites that use SEED include:

```
CipherSuite TLS_RSA_WITH_SEED_CBC_SHA      = { 0x00,
0x96};
CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA    = { 0x00,
0x97};
CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA    = { 0x00,
0x98};
CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA   = { 0x00,
0x99};
```

```

12898 CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00,
12899         0x9A};
12900 CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00,
12901         0x9B};

```

12902

12903 As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as

12904 in a MAC algorithm such as HMAC.

12905 OIDs have been published for all these uses. A list may be seen at

12906 <http://www.alvestrand.no/objectid/1.2.410.200004.1.html>

12907

12908 *Table 222, SEED Mechanisms vs. Functions*

Mechanism	Functions						
	Encryp t & Decryp t	Sign & Verif y	SR & VR 1	Diges t	Gen . Key/ Key Pair	Wrap & Unwra p	Deriv e
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							✓

12909 6.51.1 Definitions

12910 This section defines the key type “CKK_SEED” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE

12911 attribute of key objects.

12912 Mechanisms:

```

12913 CKM_SEED_KEY_GEN
12914 CKM_SEED_ECB
12915 CKM_SEED_CBC
12916 CKM_SEED_MAC
12917 CKM_SEED_MAC_GENERAL
12918 CKM_SEED_CBC_PAD

```

12919

12920 For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**

12921 are always 16.

12922 6.51.2 SEED secret key objects

12923 SEED secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_SEED**) hold SEED keys.

12924 The following table defines the secret key object attributes, in addition to the common attributes defined

12925 for this object class:

12926 Table 223, SEED Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 16 bytes long)

12927 Refer to Table 11 for footnotes

12928 The following is a sample template for creating a SEED secret key object:

```
12929 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
12930 CK_KEY_TYPE keyType = CKK_SEED;
12931 CK_UTF8CHAR label[] = "A SEED secret key object";
12932 CK_BYTE value[] = {...};
12933 CK_BBOOL true = CK_TRUE;
12934 CK_ATTRIBUTE template[] = {
12935     {CKA_CLASS, &class, sizeof(class)},
12936     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
12937     {CKA_TOKEN, &true, sizeof(true)},
12938     {CKA_LABEL, label, sizeof(label)-1},
12939     {CKA_ENCRYPT, &true, sizeof(true)},
12940     {CKA_VALUE, value, sizeof(value)}
12941 };
```

12942 **6.51.3 SEED key generation**

12943 The SEED key generation mechanism, denoted CKM_SEED_KEY_GEN, is a key generation mechanism
12944 for SEED.

12945 It does not have a parameter.

12946 The mechanism generates SEED keys.

12947 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12948 key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions
12949 the key supports) may be specified in the template for the key, or else are assigned default initial values.

12950 **6.51.4 SEED-ECB**

12951 SEED-ECB, denoted **CKM_SEED_ECB**, is a mechanism for single- and multiple-part encryption and
12952 decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

12953 It does not have a parameter.

12954 **6.51.5 SEED-CBC**

12955 SEED-CBC, denoted **CKM_SEED_CBC**, is a mechanism for single- and multiple-part encryption and
12956 decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

12957 It has a parameter, a 16-byte initialization vector.

12958 **6.51.6 SEED-CBC with PKCS padding**

12959 SEED-CBC with PKCS padding, denoted **CKM_SEED_CBC_PAD**, is a mechanism for single- and
12960 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-
12961 block chaining mode; and the block cipher padding method detailed in PKCS #7.

12962 It has a parameter, a 16-byte initialization vector.

6.51.7 General-length SEED-MAC

General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and multiple-part signatures and verification, based on SEED and data authentication.

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

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

6.51.8 SEED-MAC

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

It does not have a parameter.

6.52 Key derivation by data encryption - SEED

These mechanisms allow derivation of keys using the result of an encryption operation as the key value. They are for use with the C_DeriveKey function.

6.52.1 Definitions

Mechanisms:

CKM_SEED_ECB_ENCRYPT_DATA

CKM_SEED_CBC_ENCRYPT_DATA

```
typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS {  
    CK_BYTE      iv[16];  
    CK_BYTE_PTR  pData;  
    CK_ULONG     length;  
} CK_SEED_CBC_ENCRYPT_DATA_PARAMS;
```

```
typedef CK_SEED_CBC_ENCRYPT_DATA_PARAMS CK_PTR  
CK_SEED_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

6.52.2 Mechanism Parameters

Table 224, Mechanism Parameters for SEED-based key derivation

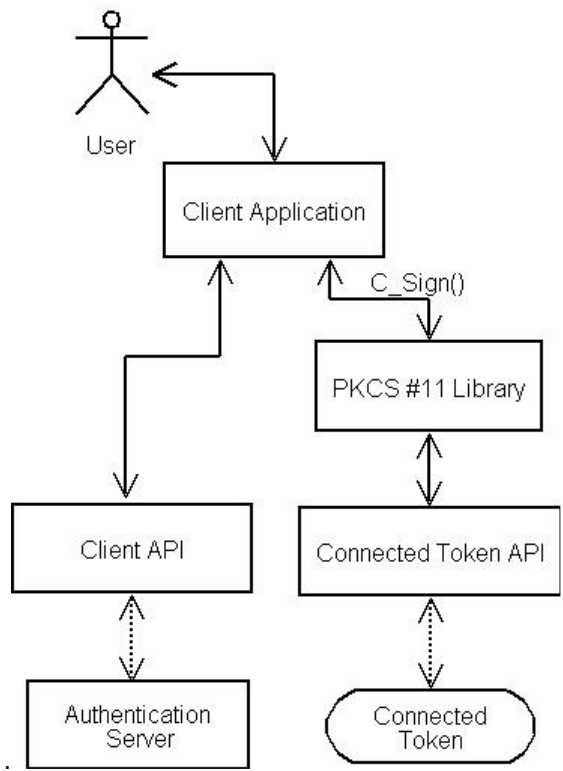
CKM_SEED_ECB_ENCRYPT_DATA	Uses CK_KEY_DERIVATION_STRING_DATA structure. Parameter is the data to be encrypted and must be a multiple of 16 long.
CKM_SEED_CBC_ENCRYPT_DATA	Uses CK_SEED_CBC_ENCRYPT_DATA_PARAMS . Parameter is an 16 byte IV value followed by the data. The data value part must be a multiple of 16 bytes long.

12993 **6.53 OTP**

12994 **6.53.1 Usage overview**

12995 OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases
12996 can be categorized according to the type of sought functionality.

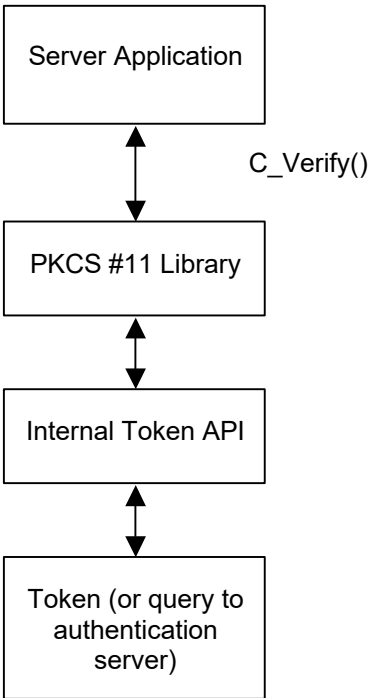
12997 **6.53.2 Case 1: Generation of OTP values**



12998
12999 *Figure 2: Retrieving OTP values through C_Sign*

13000 Figure 2 shows an integration of PKCS #11 into an application that needs to authenticate users holding
13001 OTP tokens. In this particular example, a connected hardware token is used, but a software token is
13002 equally possible. The application invokes **C_Sign** to retrieve the OTP value from the token. In the
13003 example, the application then passes the retrieved OTP value to a client API that sends it via the network
13004 to an authentication server. The client API may implement a standard authentication protocol such as
13005 RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's
13006 ACE/Agent® software.

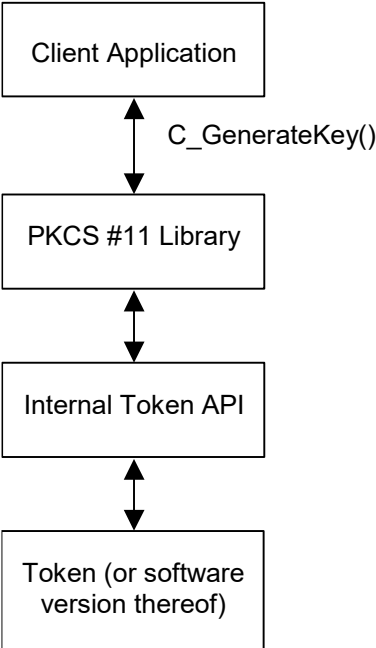
13007 **6.53.3 Case 2: Verification of provided OTP values**



13008
13009 *Figure 3: Server-side verification of OTP values*

13010 Figure 3 illustrates the server-side equivalent of the scenario depicted in Figure 2. In this case, a server
13011 application invokes **C_Verify** with the received OTP value as the signature value to be verified.

13012 **6.53.4 Case 3: Generation of OTP keys**



13014 *Figure 4: Generation of an OTP key*

13015 Figure 4 shows an integration of PKCS #11 into an application that generates OTP keys. The application

13016 invokes **C_GenerateKey** to generate an OTP key of a particular type on the token. The key may

13017 subsequently be used as a basis to generate OTP values.

13018 **6.53.5 OTP objects**

13019 **6.53.5.1 Key objects**

13020 OTP key objects (object class **CKO_OTP_KEY**) hold secret keys used by OTP tokens. The following

13021 table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys,

13022 all of which are inherited by this class:

13023 *Table 225: 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.

13024 Refer to Table 11 for footnotes

13025 Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-
13026 PKCS #11 application should therefore always consult the state of the CKA_OTP_PIN_REQUIREMENT
13027 attribute before each call to **C_SignInit**, as the value of this attribute may change dynamically.

13028 For OTP tokens with multiple keys, the keys may be enumerated using **C_FindObjects**. The
13029 **CKA_OTP_SERVICE_IDENTIFIER** and/or the **CKA_OTP_SERVICE_LOGO** attribute may be used to
13030 distinguish between keys. The actual choice of key for a particular operation is however application-
13031 specific and beyond the scope of this document.

13032 For all OTP keys, the CKA_ALLOWED_MECHANISMS attribute should be set as required.

13033 **6.53.6 OTP-related notifications**

13034 This document extends the set of defined notifications as follows:

13035 CKN_OTP_CHANGED Cryptoki is informing the application that the OTP for a key on a
13036 connected token just changed. This notification is particularly useful
13037 when applications wish to display the current OTP value for time-
13038 based mechanisms.

13039 **6.53.7 OTP mechanisms**

13040 The following table shows, for the OTP mechanisms defined in this document, their support by different
13041 cryptographic operations. For any particular token, of course, a particular operation may well support
13042 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
13043 mechanism for some operation supports any other mechanism for any other operation (or even supports
13044 that same mechanism for any other operation).

13045 *Table 226: 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		✓					

13046 The remainder of this section will present in detail the OTP mechanisms and the parameters that are
13047 supplied to them.

13048 **6.53.7.1 OTP mechanism parameters**

13049 ♦ **CK_OTP_PARAM_TYPE**

13050 **CK_OTP_PARAM_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

13051 typedef CK_ULONG CK_OTP_PARAM_TYPE;

13052 The following **CK_OTP_PARAM_TYPE** types are defined:

13053 *Table 227, 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_OUTPUT_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.

13054

13055 The following table defines the possible values for the CK_OTP_FLAGS type:

13056 Table 228: 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.

Note: Even if CKA_OTP_FORMAT is not set to CK_OTP_FORMAT_BINARY, then there may still be value in setting the CKF_USER_FRIENDLY_OTP flag (assuming CKA_OTP_USER_FRIENDLY_MODE is CK_TRUE, of course) if the intent is for a human to read the generated OTP value, since it may become shorter or otherwise better suited for a user. Applications that do not intend to provide a returned OTP value to a user should not set the CKF_USER_FRIENDLY_OTP flag.

◆ CK_OTP_PARAM; CK_OTP_PARAM_PTR

CK_OTP_PARAM is a structure that includes the type, value, and length of an OTP parameter. It is defined as follows:

```

13065     typedef struct CK_OTP_PARAM {
13066         CK_OTP_PARAM_TYPE type;
13067         CK_VOID_PTR pValue;
13068         CK_ULONG ulValueLen;
13069     } CK_OTP_PARAM;

```

13070 The fields of the structure have the following meanings:

13071	type	the parameter type
13072	pValue	pointer to the value of the parameter
13073	ulValueLen	length in bytes of the value

13074 If a parameter has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. Note that *pValue*
13075 is a “void” pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library
13076 must ensure that the pointer can be safely cast to the expected type (*i.e.*, without word-alignment errors).

13077 **CK_OTP_PARAM_PTR** is a pointer to a **CK_OTP_PARAM**.

13078

13079 ♦ **CK_OTP_PARAMS; CK_OTP_PARAMS_PTR**

13080 **CK_OTP_PARAMS** is a structure that is used to provide parameters for OTP mechanisms in a generic
13081 fashion. It is defined as follows:

```

13082     typedef struct CK_OTP_PARAMS {
13083         CK_OTP_PARAM_PTR pParams;
13084         CK_ULONG ulCount;
13085     } CK_OTP_PARAMS;

```

13086 The fields of the structure have the following meanings:

13087	pParams	pointer to an array of OTP parameters
13088	ulCount	the number of parameters in the array

13089 **CK_OTP_PARAMS_PTR** is a pointer to a **CK_OTP_PARAMS**.

13090

13091 When calling *C_SignInit* or *C_VerifyInit* with a mechanism that takes a **CK_OTP_PARAMS** structure as a
13092 parameter, the **CK_OTP_PARAMS** structure shall be populated in accordance with the
13093 **CKA_OTP_X_REQUIREMENT** key attributes for the identified key, where *X* is PIN, CHALLENGE, TIME,
13094 or COUNTER.

13095 For example, if **CKA_OTP_TIME_REQUIREMENT** = **CK_OTP_PARAM_MANDATORY**, then the
13096 **CK_OTP_TIME** parameter shall be present. If **CKA_OTP_TIME_REQUIREMENT** =
13097 **CK_OTP_PARAM_OPTIONAL**, then a **CK_OTP_TIME** parameter may be present. If it is not present,
13098 then the library may collect it (during the *C_Sign* call). If **CKA_OTP_TIME_REQUIREMENT** =
13099 **CK_OTP_PARAM_IGNORED**, then a provided **CK_OTP_TIME** parameter will always be ignored.
13100 Additionally, a provided **CK_OTP_TIME** parameter will always be ignored if **CKF_EXCLUDE_TIME** is set
13101 in a **CK_OTP_FLAGS** parameter. Similarly, if this flag is set, a library will not attempt to collect the value
13102 itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is
13103 an error (**CKR_MECHANISM_PARAM_INVALID**) to set the **CKF_EXCLUDE_TIME** flag when the
13104 **CKA_OTP_TIME_REQUIREMENT** attribute is **CK_OTP_PARAM_MANDATORY**.

13105 The above discussion holds for all **CKA_OTP_X_REQUIREMENT** attributes (*i.e.*,
13106 **CKA_OTP_PIN_REQUIREMENT**, **CKA_OTP_CHALLENGE_REQUIREMENT**,
13107 **CKA_OTP_COUNTER_REQUIREMENT**, **CKA_OTP_TIME_REQUIREMENT**). A library may set a
13108 particular **CKA_OTP_X_REQUIREMENT** attribute to **CK_OTP_PARAM_OPTIONAL** even if it is required
13109 by the mechanism as long as the token (or the library itself) has the capability of providing the value to the
13110 computation. One example of this is a token with an on-board clock.

In addition, applications may use the CK_OTP_FLAGS, the CK_OTP_OUTPUT_FORMAT and the CKA_OTP_LENGTH parameters to set additional parameters.

◆ CK_OTP_SIGNATURE_INFO, CK_OTP_SIGNATURE_INFO_PTR

CK_OTP_SIGNATURE_INFO is a structure that is returned by all OTP mechanisms in successful calls to **C_Sign** (**C_SignFinal**). The structure informs applications of actual parameter values used in particular OTP computations in addition to the OTP value itself. It is used by all mechanisms for which the key belongs to the class **CKO_OTP_KEY** and is defined as follows:

```
typedef struct CK_OTP_SIGNATURE_INFO {  
    CK_OTP_PARAM_PTR pParams;  
    CK_ULONG ulCount;  
} CK_OTP_SIGNATURE_INFO;
```

The fields of the structure have the following meanings:

pParams	pointer to an array of OTP parameter values
ulCount	the number of parameters in the array

After successful calls to **C_Sign** or **C_SignFinal** with an OTP mechanism, the *pSignature* parameter will be set to point to a **CK_OTP_SIGNATURE_INFO** structure. One of the parameters in this structure will be the OTP value itself, identified with the **CK_OTP_VALUE** tag. Other parameters may be present for informational purposes, e.g. the actual time used in the OTP calculation. In order to simplify OTP validations, authentication protocols may permit authenticating parties to send some or all of these parameters in addition to OTP values themselves. Applications should therefore check for their presence in returned **CK_OTP_SIGNATURE_INFO** values whenever such circumstances apply.

Since **C_Sign** and **C_SignFinal** follows the convention described in Section 5.2 on producing output, a call to **C_Sign** (or **C_SignFinal**) with *pSignature* set to **NULL_PTR** will return (in the *pulSignatureLen* parameter) the required number of bytes to hold the **CK_OTP_SIGNATURE_INFO** structure as well as all the data in all its **CK_OTP_PARAM** components. If an application allocates a memory block based on this information, it shall therefore not subsequently de-allocate components of such a received value but rather de-allocate the complete **CK_OTP_PARAMS** structure itself. A Cryptoki library that is called with a non-NULL *pSignature* pointer will assume that it points to a *contiguous* memory block of the size indicated by the *pulSignatureLen* parameter.

When verifying an OTP value using an OTP mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure returned by a call to **C_Sign**. The **CK_OTP_PARAM** value supplied in the **C_VerifyInit** call sets the values to use in the verification operation.

CK_OTP_SIGNATURE_INFO_PTR points to a **CK_OTP_SIGNATURE_INFO**.

6.53.8 RSA SecurID

6.53.8.1 RSA SecurID secret key objects

RSA SecurID secret key objects (object class **CKO_OTP_KEY**, key type **CKK_SECURID**) hold RSA SecurID secret keys. The following table defines the RSA SecurID secret key object attributes, in addition to the common attributes defined for this object class:

Table 229, 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.

¹ Refer to Table 11 for footnotes

The following is a sample template for creating an RSA SecurID secret key object:

```

13154     CK_OBJECT_CLASS class = CKO_OTP_KEY;
13155     CK_KEY_TYPE keyType = CKK_SECURID;
13156     CK_DATE endDate = {...};
13157     CK_UTF8CHAR label[] = "RSA SecurID secret key object";
13158     CK_BYTE keyId[] = {...};
13159     CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
13160     CK_ULONG outputLength = 6;
13161     CK_ULONG needPIN = CK_OTP_PARAM_MANDATORY;
13162     CK_ULONG timeInterval = 60;
13163     CK_BYTE value[] = {...};
13164     CK_BBOOL true = CK_TRUE;
13165     CK_ATTRIBUTE template[] = {
13166         {CKA_CLASS, &class, sizeof(class)},
13167         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13168         {CKA_END_DATE, &endDate, sizeof(endDate)},
13169         {CKA_TOKEN, &true, sizeof(true)},
13170         {CKA_SENSITIVE, &true, sizeof(true)},
13171         {CKA_LABEL, label, sizeof(label)-1},
13172         {CKA_SIGN, &true, sizeof(true)},
13173         {CKA_VERIFY, &true, sizeof(true)},
13174         {CKA_ID, keyId, sizeof(keyId)},
13175         {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
13176         {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
13177         {CKA_OTP_PIN_REQUIREMENT, &needPIN, sizeof(needPIN)},
13178         {CKA_OTP_TIME_INTERVAL, &timeInterval,
13179             sizeof(timeInterval)},
13180         {CKA_VALUE, value, sizeof(value)}
13181     };

```

13182 6.53.8.2 RSA SecurID key generation

13183 The RSA SecurID key generation mechanism, denoted **CKM_SECURID_KEY_GEN**, is a key generation
13184 mechanism for the RSA SecurID algorithm.

13185 It does not have a parameter.

13186 The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the
13187 template for the key.

13188 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE_LEN**, and
13189 **CKA_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be
13190 specified in the template for the key, or else are assigned default initial values

13191 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13192 specify the supported range of SecurID key sizes, in bytes.

13193 6.53.8.3 SecurID OTP generation and validation

13194 **CKM_SECURID** is the mechanism for the retrieval and verification of RSA SecurID OTP values.

13195 The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

13196 When signing or verifying using the **CKM_SECURID** mechanism, *pData* shall be set to **NULL_PTR** and
13197 *ulDataLen* shall be set to 0.

6.53.8.4 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.

6.53.9 OATH HOTP

6.53.9.1 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:

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_HOTP;
CK_UTF8CHAR label[] = "HOTP secret key object";
CK_BYTE keyId[] = {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_END_DATE, &endDate, sizeof(endDate)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VERIFY, &true, sizeof(true)},
    {CKA_ID, keyId, sizeof(keyId)},
    {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
    {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
    {CKA_OTP_COUNTER, counterValue, sizeof(counterValue)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.53.9.2 HOTP key generation

The HOTP key generation mechanism, denoted **CKM_HOTP_KEY_GEN**, is a key generation mechanism for the HOTP algorithm.

It does not have a parameter.

The mechanism generates HOTP keys with a particular set of attributes as specified in the template for the key.

The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_OTP_COUNTER**, **CKA_VALUE** and **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the HOTP key type may be specified in the template for the key, or else are assigned default initial values.

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

6.53.9.3 HOTP OTP generation and validation

CKM_HOTP is the mechanism for the retrieval and verification of HOTP OTP values based on the current internal counter, or a provided counter.

The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism, *pData* shall be set to **NULL_PTR** and *ulDataLen* shall be set to 0.

For verify operations, the counter value **CK_OTP_COUNTER** must be provided as a **CK_OTP_PARAM** parameter to **C_VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, *pSignature* shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure in the case of an earlier call to **C_Sign**.

6.53.10 ActivIdentity ACTI

6.53.10.1 ACTI secret key objects

ACTI secret key objects (object class **CKO_OTP_KEY**, key type **CKK_ACTI**) hold ActivIdentity ACTI secret keys.

For ACTI 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 the **CK_OTP_COUNTER** value in the **CK_OTP_PARAM** structure.

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**.

The **CKA_OTP_TIME** 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**.

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

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_ACTI;
CK_UTF8CHAR label[] = "ACTI secret key object";
CK_BYTE keyId[] = {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_DATE endDate = {...};
CK_BYTE counterValue[8] = {0};
```



```

13288     CK_BYTE value[] = {...};
13289     CK_BBOOL true = CK_TRUE;
13290     CK_ATTRIBUTE template[] = {
13291         {CKA_CLASS, &class, sizeof(class)},
13292         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13293         {CKA_END_DATE, &endDate, sizeof(endDate)},
13294         {CKA_TOKEN, &true, sizeof(true)},
13295         {CKA_SENSITIVE, &true, sizeof(true)},
13296         {CKA_LABEL, label, sizeof(label)-1},
13297         {CKA_SIGN, &true, sizeof(true)},
13298         {CKA_VERIFY, &true, sizeof(true)},
13299         {CKA_ID, keyId, sizeof(keyId)},
13300         {CKA_OTP_FORMAT, &outputFormat,
13301         sizeof(outputFormat)},
13302         {CKA_OTP_LENGTH, &outputLength,
13303         sizeof(outputLength)},
13304         {CKA_OTP_COUNTER, counterValue,
13305         sizeof(counterValue)},
13306         {CKA_VALUE, value, sizeof(value)}
13307     };

```

13308 6.53.10.2 ACTI key generation

13309 The ACTI key generation mechanism, denoted **CKM_ACTI_KEY_GEN**, is a key generation mechanism
13310 for the ACTI algorithm.

13311 It does not have a parameter.

13312 The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the
13313 key.

13314 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE** and
13315 **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be
13316 specified in the template for the key, or else are assigned default initial values.

13317 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13318 specify the supported range of ACTI key sizes, in bytes.

13319 6.53.10.3 ACTI OTP generation and validation

13320 **CKM_ACTI** is the mechanism for the retrieval and verification of ACTI OTP values.

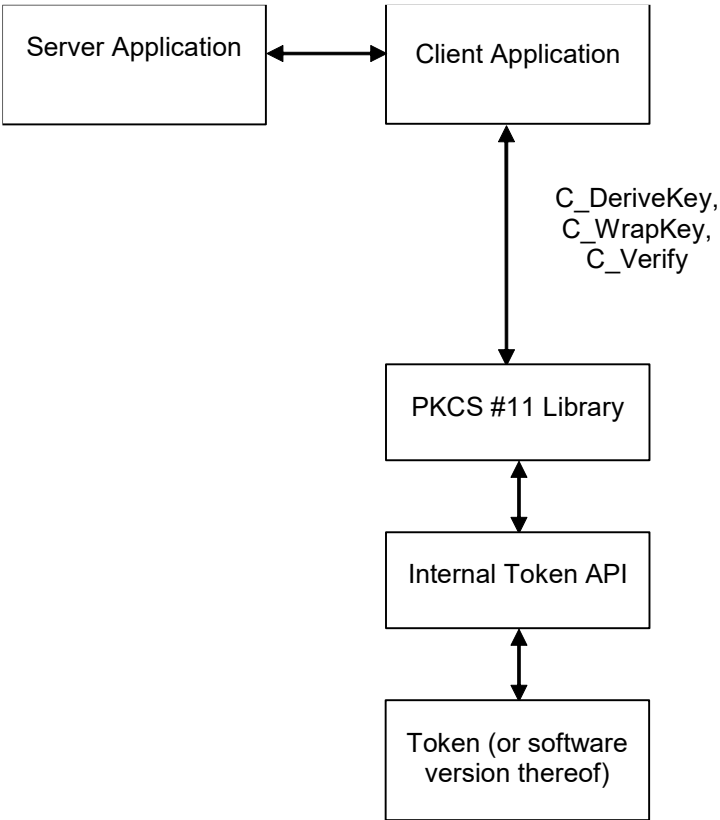
13321 The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

13322 When signing or verifying using the **CKM_ACTI** mechanism, *pData* shall be set to **NULL_PTR** and
13323 *ulDataLen* shall be set to 0.

13324 When verifying an OTP value using the **CKM_ACTI** mechanism, *pSignature* shall be set to the OTP value
13325 itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure in the case of
13326 an earlier call to **C_Sign**.

13327 **6.54 CT-KIP**

13328 **6.54.1 Principles of Operation**



13329
13330 *Figure 5: PKCS #11 and CT-KIP integration*

13331 Figure 5 shows an integration of PKCS #11 into an application that generates cryptographic keys through
13332 the use of CT-KIP. The application invokes **C_DeriveKey** to derive a key of a particular type on the token.
13333 The key may subsequently be used as a basis to e.g., generate one-time password values. The
13334 application communicates with a CT-KIP server that participates in the key derivation and stores a copy
13335 of the key in its database. The key is transferred to the server in wrapped form, after a call to
13336 **C_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls
13337 to **C_Verify**.

13338 **6.54.2 Mechanisms**

13339 The following table shows, for the mechanisms defined in this document, their support by different
13340 cryptographic operations. For any particular token, of course, a particular operation may well support
13341 only a subset of the mechanisms listed. There is also no guarantee that a token that supports one
13342 mechanism for some operation supports any other mechanism for any other operation (or even supports
13343 that same mechanism for any other operation).

13344 *Table 230: 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		✓					

The remainder of this section will present in detail the mechanisms and the parameters that are supplied to them.

6.54.3 Definitions

Mechanisms:

CKM_KIP_DERIVE

CKM_KIP_WRAP

CKM_KIP_MAC

6.54.4 CT-KIP Mechanism parameters

◆ CK_KIP_PARAMS; CK_KIP_PARAMS_PTR

CK_KIP_PARAMS is a structure that provides the parameters to all the CT-KIP related mechanisms: The **CKM_KIP_DERIVE** key derivation mechanism, the **CKM_KIP_WRAP** key wrap and key unwrap mechanism, and the **CKM_KIP_MAC** signature mechanism. The structure is defined as follows:

```
typedef struct CK_KIP_PARAMS {
    CK_MECHANISM_PTR    pMechanism;
    CK_OBJECT_HANDLE    hKey;
    CK_BYTE_PTR         pSeed;
    CK_ULONG            ulSeedLen;
} CK_KIP_PARAMS;
```

The fields of the structure have the following meanings:

pMechanism	pointer to the underlying cryptographic mechanism (e.g. AES, SHA-256)
hKey	handle to a key that will contribute to the entropy of the derived key (CKM_KIP_DERIVE) or will be used in the MAC operation (CKM_KIP_MAC)
pSeed	pointer to an input seed
ulSeedLen	length in bytes of the input seed

CK_KIP_PARAMS_PTR is a pointer to a **CK_KIP_PARAMS** structure.

6.54.5 CT-KIP key derivation

The CT-KIP key derivation mechanism, denoted **CKM_KIP_DERIVE**, is a key derivation mechanism that is capable of generating secret keys of potentially any type, subject to token limitations.

It takes a parameter of type **CK_KIP_PARAMS** which allows for the passing of the desired underlying cryptographic mechanism as well as some other data. In particular, when the *hKey* parameter is a handle

13377 to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C_DeriveKey**.
13378 The *pSeed* parameter may be used to seed the key derivation operation.
13379 The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the
13380 template for the key.
13381 The mechanism contributes the **CKA_CLASS** and **CKA_VALUE** attributes to the new key. Other
13382 attributes supported by the key type may be specified in the template for the key, or else will be assigned
13383 default initial values. Since the mechanism is generic, the **CKA_KEY_TYPE** attribute should be set in the
13384 template, if the key is to be used with a particular mechanism.

13385 **6.54.6 CT-KIP key wrap and key unwrap**

13386 The CT-KIP key wrap and unwrap mechanism, denoted **CKM_KIP_WRAP**, is a key wrap mechanism that
13387 is capable of wrapping and unwrapping generic secret keys.
13388 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
13389 cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of
13390 **CK_KIP_PARAMS**.

13391 **6.54.7 CT-KIP signature generation**

13392 The CT-KIP signature (MAC) mechanism, denoted **CKM_KIP_MAC**, is a mechanism used to produce a
13393 message authentication code of arbitrary length. The keys it uses are secret keys.
13394 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
13395 cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed*
13396 and the *ulSeedLen* parameters of **CT_KIP_PARAMS**.
13397 This mechanism produces a MAC of the length specified by *puSignatureLen* parameter in calls to
13398 **C_Sign**.
13399 If a call to **C_Sign** with this mechanism fails, then no output will be generated.

13400 **6.55 GOST 28147-89**

13401 GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

13402

13403 *Table 231, 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						✓	

13404

13405 **6.55.1 Definitions**

13406 This section defines the key type “CKK_GOST28147” for type **CK_KEY_TYPE** as used in the
13407 **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

- 13408 Mechanisms:
- 13409 CKM_GOST28147_KEY_GEN
- 13410 CKM_GOST28147_ECB
- 13411 CKM_GOST28147
- 13412 CKM_GOST28147_MAC
- 13413 CKM_GOST28147_KEY_WRAP

13414

6.55.2 GOST 28147-89 secret key objects

13415 GOST 28147-89 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_GOST28147**) hold

13416 GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in

13417 addition to the common attributes defined for this object class:

13418

Table 232, 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

13419

Refer to Table 11 for footnotes

13420

The following is a sample template for creating a GOST 28147-89 secret key object:

13421 CK_OBJECT_CLASS class = CKO_SECRET_KEY;

13422 CK_KEY_TYPE keyType = CKK_GOST28147;

13423 CK_UTF8CHAR label[] = "A GOST 28147-89 secret key object";

13424 CK_BYTE value[32] = {...};

13425 CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,

13426 0x02, 0x1f, 0x00};

13427 CK_BBOOL true = CK_TRUE;

13428 CK_ATTRIBUTE template[] = {

13429 {CKA_CLASS, &class, sizeof(class)},

13430 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},

13431 {CKA_TOKEN, &>true, sizeof(true)},

13432 {CKA_LABEL, label, sizeof(label)-1},

13433 {CKA_ENCRYPT, &true, sizeof(true)},

13434 {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},

13435 {CKA_VALUE, value, sizeof(value)}

13436 };

13437

6.55.3 GOST 28147-89 domain parameter objects

13438 GOST 28147-89 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type

13439 **CKK_GOST28147**) hold GOST 28147-89 domain parameters.

13440 The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the

13441 common attributes defined for this object class:

Table 233, 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

¹ Refer to 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 28147-89 domain parameter object:

```

CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOST28147;
CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic
    parameters object";
CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,
    0x1f, 0x00};
CK_BYTE value[] = {
    0x30, 0x62, 0x04, 0x40, 0x4c, 0xde, 0x38, 0x9c, 0x29, 0x89, 0xef, 0xb6,
    0xff, 0xeb, 0x56, 0xc5, 0x5e, 0xc2, 0x9b, 0x02, 0x98, 0x75, 0x61, 0x3b,
    0x11, 0x3f, 0x89, 0x60, 0x03, 0x97, 0x0c, 0x79, 0x8a, 0xa1, 0xd5, 0x5d,
    0xe2, 0x10, 0xad, 0x43, 0x37, 0x5d, 0xb3, 0x8e, 0xb4, 0x2c, 0x77, 0xe7,
    0xcd, 0x46, 0xca, 0xfa, 0xd6, 0x6a, 0x20, 0x1f, 0x70, 0xf4, 0x1e, 0xa4,
    0xab, 0x03, 0xf2, 0x21, 0x65, 0xb8, 0x44, 0xd8, 0x02, 0x01, 0x00, 0x02,
    0x01, 0x40, 0x30, 0x0b, 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x0e,
    0x00, 0x05, 0x00
};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};

```

6.55.4 GOST 28147-89 key generation

The GOST 28147-89 key generation mechanism, denoted **CKM_GOST28147_KEY_GEN**, is a key generation mechanism for GOST 28147-89.

It does not have a parameter.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object class **CKO_SECRET_KEY**.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are not used.

6.55.5 GOST 28147-89-ECB

GOST 28147-89-ECB, denoted **CKM_GOST28147_ECB**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on GOST 28147-89 and electronic codebook mode.

It does not have a parameter.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size so that the resulting length is a multiple of the block size.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, 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 mechanism contributes the result as the **CKA_VALUE** attribute of the new key.

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

Table 234, 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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.55.6 GOST 28147-89 encryption mode except ECB

GOST 28147-89 encryption mode except ECB, denoted **CKM_GOST28147**, is a mechanism for single and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on [GOST 28147-89] and CFB, counter mode, and additional CBC mode defined in [RFC 4357] section 2. Encryption's parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

It has a parameter, which is an 8-byte initialization vector. This parameter may be omitted then a zero initialization vector is used.

This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to wrap/unwrap every secret key that it supports.

For wrapping (**C_WrapKey**), the mechanism encrypts the value of the **CKA_VALUE** attribute of the key that is wrapped.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA_VALUE** attribute of the new key.

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

Table 235, 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	

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.55.7 GOST 28147-89-MAC

GOST 28147-89-MAC, denoted **CKM_GOST28147_MAC**, is a mechanism for data integrity and authentication based on GOST 28147-89 and key meshing algorithms [RFC 4357] section 2.3.

MACing parameters are specified in object identifier of attribute **CKA_GOST28147_PARAMS**.

The output bytes from this mechanism are taken from the start of the final GOST 28147-89 cipher block produced in the MACing process.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

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

Table 236, 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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89

GOST 28147-89 keys as a KEK (key encryption keys) for encryption GOST 28147-89 keys, denoted by **CKM_GOST28147_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST 28147-89. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89.

For wrapping (**C_WrapKey**), the mechanism first computes MAC from the value of the **CKA_VALUE** attribute of the key that is wrapped and then encrypts in ECB mode the value of the **CKA_VALUE** attribute of the key that is wrapped. The result is 32 bytes of the key that is wrapped and 4 bytes of MAC.

For unwrapping (**C_UnwrapKey**), the mechanism first decrypts in ECB mode the 32 bytes of the key that was wrapped and then computes MAC from the unwrapped key. Then compared together 4 bytes MAC has computed and 4 bytes MAC of the input. If these two MACs do not match the wrapped key is disallowed. The mechanism contributes the result as the **CKA_VALUE** attribute of the unwrapped key.

It has a parameter, which is an 8-byte MAC initialization vector. This parameter may be omitted then a zero initialization vector is used.

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

Table 237, 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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.56 GOST R 34.11-94

GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit message digest defined in [GOST R 34.11-94].

Table 238, 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		✓					

6.56.1 Definitions

This section defines the key type “CKK_GOSTR3411” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of domain parameter objects.

Mechanisms:

CKM_GOSTR3411

CKM_GOSTR3411_HMAC

6.56.2 GOST R 34.11-94 domain parameter objects

GOST R 34.11-94 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_GOSTR3411**) hold GOST R 34.11-94 domain parameters.

The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the common attributes defined for this object class:

Table 239, 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

¹ Refer to Table 11 for footnotes

6.56.4 GOST R 34.11-94 HMAC

GOST R 34.11-94 HMAC mechanism, denoted **CKM_GOSTR3411_HMAC**, is a mechanism for signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash function [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key type **CKK_GENERIC_SECRET** or **CKK_GOST28147**.

To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357] (section 11.2) must be used.

Signatures (MACs) produced by this mechanism are of 32 bytes long.

Constraints on the length of input and output data are summarized in the following table:

Table 241, 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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.57 GOST R 34.10-2001

GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following the digital signature algorithm defined in [GOST R 34.10-2001].

Table 242, GOST R34.10-2001 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	S R & V R	Digest	Gen . Key/ Key Pair	Wrap & Unwrap	Derive
CKM_GOSTR3410_KEY_PAIR_GEN					✓		
CKM_GOSTR3410		✓ ¹					
CKM_GOSTR3410_WITH_GOSTR3411		✓					
CKM_GOSTR3410_KEY_WRAP						✓	
CKM_GOSTR3410_DERIVE							✓

¹ Single-part operations only

6.57.1 Definitions

This section defines the key type “CKM_GOSTR3410” for type CK_KEY_TYPE as used in the CK_KEY_TYPE attribute of key objects and domain parameter objects.

Mechanisms:

13640 CKM_GOSTR3410_KEY_PAIR_GEN
13641 CKM_GOSTR3410
13642 CKM_GOSTR3410_WITH_GOSTR3411
13643 CKM_GOSTR3410
13644 CKM_GOSTR3410_KEY_WRAP
13645 CKM_GOSTR3410_DERIVE

13646 **6.57.2 GOST R 34.10-2001 public key objects**

13647 GOST R 34.10-2001 public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_GOSTR3410**)
13648 hold GOST R 34.10-2001 public keys.

13649 The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the
13650 common attributes defined for this object class:

13651 *Table 243, 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

13652 Refer to Table 11 for footnotes

13653 The following is a sample template for creating an GOST R 34.10-2001 public key object:

```
13654 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
13655 CK_KEY_TYPE keyType = CKK_GOSTR3410;  
13656 CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";  
13657 CK_BYTE gostR3410params_oid[] =  
13658     {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};  
13659 CK_BYTE gostR3411params_oid[] =
```

```

13660     {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
13661     CK_BYTE gost28147params_oid[] =
13662     {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
13663     CK_BYTE value[64] = {...};
13664     CK_BBOOL true = CK_TRUE;
13665     CK_ATTRIBUTE template[] = {
13666         {CKA_CLASS, &class, sizeof(class)},
13667         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
13668         {CKA_TOKEN, &true, sizeof(true)},
13669         {CKA_LABEL, label, sizeof(label)-1},
13670         {CKA_GOSTR3410_PARAMS, gostR3410params_oid,
13671             sizeof(gostR3410params_oid)},
13672         {CKA_GOSTR3411_PARAMS, gostR3411params_oid,
13673             sizeof(gostR3411params_oid)},
13674         {CKA_GOST28147_PARAMS, gost28147params_oid,
13675             sizeof(gost28147params_oid)},
13676         {CKA_VALUE, value, sizeof(value)}
13677     };

```

13678 6.57.3 GOST R 34.10-2001 private key objects

13679 GOST R 34.10-2001 private key objects (object class **CKO_PRIVATE_KEY**, key type
13680 **CKK_GOSTR3410**) hold GOST R 34.10-2001 private keys.

13681 The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the
13682 common attributes defined for this object class:

13683 *Table 244, 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.

Attribute	Data Type	Meaning
		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

Refer to Table 11 for footnotes

Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain parameters are *not* specified in the key's template. This is because GOST R 34.10-2001 private keys are only generated as part of an GOST R 34.10-2001 key *pair*, and the GOST R 34.10-2001 domain parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

The following is a sample template for creating an GOST R 34.10-2001 private key object:

```

CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key
    object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_BYTE gostR3410params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE gostR3411params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
CK_BYTE gost28147params_oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_GOSTR3410_PARAMS, gostR3410params_oid,
        sizeof(gostR3410params_oid)},
    {CKA_GOSTR3411_PARAMS, gostR3411params_oid,
        sizeof(gostR3411params_oid)},
    {CKA_GOST28147_PARAMS, gost28147params_oid,
        sizeof(gost28147params_oid)},
    {CKA_VALUE, value, sizeof(value)}
};

```


6.57.4 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 245, 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

¹ Refer to 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:

```
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_GOSTR3410;
CK_UTF8CHAR label[] = "A GOST R34.10-2001 cryptographic
    parameters object";
CK_BYTE oid[] =
    {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
CK_BYTE value[] = {
    0x30, 0x81, 0x90, 0x02, 0x01, 0x07, 0x02, 0x20, 0x5f, 0xbf, 0xf4, 0x98,
    0xaa, 0x93, 0x8c, 0xe7, 0x39, 0xb8, 0xe0, 0x22, 0xfb, 0xaf, 0xef, 0x40,
    0x56, 0x3f, 0x6e, 0x6a, 0x34, 0x72, 0xfc, 0x2a, 0x51, 0x4c, 0x0c, 0xe9,
    0xda, 0xe2, 0x3b, 0x7e, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x04, 0x31, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x50, 0xfe,
    0x8a, 0x18, 0x92, 0x97, 0x61, 0x54, 0xc5, 0x9c, 0xfc, 0x19, 0x3a, 0xcc,
    0xf5, 0xb3, 0x02, 0x01, 0x02, 0x02, 0x20, 0x08, 0xe2, 0xa8, 0xa0, 0xe6,
    0x51, 0x47, 0xd4, 0xbd, 0x63, 0x16, 0x03, 0x0e, 0x16, 0xd1, 0x9c, 0x85,
    0xc9, 0x7f, 0x0a, 0x9c, 0xa2, 0x67, 0x12, 0x2b, 0x96, 0xab, 0xbc, 0xea,
    0x7e, 0x8f, 0xc8
};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_OBJECT_ID, oid, sizeof(oid)},
    {CKA_VALUE, value, sizeof(value)}
};
```

13763

13764 **6.57.5 GOST R 34.10-2001 mechanism parameters**

13765 **♦ CK_GOSTR3410_KEY_WRAP_PARAMS**

13766 **CK_GOSTR3410_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
13767 **CKM_GOSTR3410_KEY_WRAP** mechanism. It is defined as follows:

```
13768     typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {  
13769         CK_BYTE_PTR      pWrapOID;  
13770         CK_ULONG         ulWrapOIDLen;  
13771         CK_BYTE_PTR      pUKM;  
13772         CK_ULONG         ulUKMLen;  
13773         CK_OBJECT_HANDLE hKey;  
13774     } CK_GOSTR3410_KEY_WRAP_PARAMS;
```

13775

13776 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 NULL_PTR then equal to 8
<i>hKey</i>	key handle. Key handle of a sender for C_WrapKey operation. Key handle of a receiver for C_UnwrapKey operation. When key handle takes CK_INVALID_HANDLE value then an ephemeral (one time) key pair of a sender will be used

13777 CK_GOSTR3410_KEY_WRAP_PARAMS_PTR is a pointer to a
13778 CK_GOSTR3410_KEY_WRAP_PARAMS.

13779 **♦ CK_GOSTR3410_DERIVE_PARAMS**

13780 **CK_GOSTR3410_DERIVE_PARAMS** is a structure that provides the parameters to the
13781 **CKM_GOSTR3410_DERIVE** mechanism. It is defined as follows:

```
13782     typedef struct CK_GOSTR3410_DERIVE_PARAMS {  
13783         CK_EC_KDF_TYPE kdf;  
13784         CK_BYTE_PTR    pPublicData;  
13785         CK_ULONG       ulPublicDataLen;
```

```

13786         CK_BYTE_PTR      pUKM;
13787         CK_ULONG          ulUKMLen;
13788     } CK_GOSTR3410_DERIVE_PARAMS;

```

13789

13790 The fields of the structure have the following meanings:

kdf additional key diversification algorithm identifier.
Possible values are CKD_NULL and CKD_CPDIVERSIFY_KDF. In case of CKD_NULL, result of the key derivation function described in [RFC 4357], section 5.2 is used directly; In case of CKD_CPDIVERSIFY_KDF, the resulting key value is additionally processed with algorithm from [RFC 4357], section 6.5.

*pPublicData*¹ pointer to data with public key of a receiver

ulPublicDataLen length of data with public key of a receiver (must be 64)

pUKM pointer to a UKM data

ulUKMLen length of UKM data in bytes (must be 8)

13791

13792 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
13793 them is 32 bytes long and represented in little endian order.

13794 CK_GOSTR3410_DERIVE_PARAMS_PTR is a pointer to a CK_GOSTR3410_DERIVE_PARAMS.

13795 6.57.6 GOST R 34.10-2001 key pair generation

13796 The GOST R 34.10-2001 key pair generation mechanism, denoted
13797 **CKM_GOSTR3410_KEY_PAIR_GEN**, is a key pair generation mechanism for GOST R 34.10-2001.

13798 This mechanism does not have a parameter.

13799 The mechanism generates GOST R 34.10-2001 public/private key pairs with particular
13800 GOST R 34.10-2001 domain parameters, as specified in the **CKA_GOSTR3410_PARAMS**,
13801 **CKA_GOSTR3411_PARAMS**, and **CKA_GOST28147_PARAMS** attributes of the template for the public
13802 key. Note that **CKA_GOST28147_PARAMS** attribute may not be present in the template.

13803 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
13804 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_GOSTR3410_PARAMS**,
13805 **CKA_GOSTR3411_PARAMS**, **CKA_GOST28147_PARAMS** attributes to the new private key.

13806 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13807 are not used.

13808 6.57.7 GOST R 34.10-2001 without hashing

13809 The GOST R 34.10-2001 without hashing mechanism, denoted **CKM_GOSTR3410**, is a mechanism for
13810 single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the
13811 part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

13812 This mechanism does not have a parameter.

13813 For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes
13814 long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*,
13815 both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC
13816 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is an octet string of 32 bytes long with digest has computed by means of GOST R 34.11-94 hash algorithm in the context of signed or should be signed message.

Table 246, 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

¹ Single-part operations only.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.57.8 GOST R 34.10-2001 with GOST R 34.11-94

The GOST R 34.10-2001 with GOST R 34.11-94, denoted **CKM_GOSTR3410_WITH_GOSTR3411**, is a mechanism for signatures and verification for GOST R 34.10-2001. This mechanism computes the entire GOST R 34.10-2001 specification, including the hashing with GOST R 34.11-94 hash algorithm.

As a parameter this mechanism utilizes a DER-encoding of the object identifier indicating GOST R 34.11-94 data object type. A mechanism parameter may be missed then parameters are specified in object identifier of attribute **CKA_GOSTR3411_PARAMS** must be used.

For the purposes of these mechanisms, a GOST R 34.10-2001 signature is an octet string of 64 bytes long. The signature octets correspond to the concatenation of the GOST R 34.10-2001 values *s* and *r'*, both represented as a 32 bytes octet string in big endian order with the most significant byte first [RFC 4490] section 3.2, and [RFC 4491] section 2.2.2.

The input for the mechanism is signed or should be signed message of any length. Single- and multiple-part signature operations are available.

Table 247, 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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001

GOST R 34.10-2001 keys as a KEK (key encryption keys) for encryption GOST 28147 keys, denoted by **CKM_GOSTR3410_KEY_WRAP**, is a mechanism for key wrapping; and key unwrapping, based on GOST R 34.10-2001. Its purpose is to encrypt and decrypt keys have been generated by key generation mechanism for GOST 28147-89. An encryption algorithm from [RFC 4490] (section 5.2) must be used. Encrypted key is a DER-encoded structure of ASN.1 *GostR3410-KeyTransport* type [RFC 4490] section 4.2.

It has a parameter, a **CK_GOSTR3410_KEY_WRAP_PARAMS** structure defined in section 6.57.5.

For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as the **CKA_VALUE** attribute of the new key.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used.

6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys

Common key derivation, denoted **CKM_GOSTR3410_DERIVE**, is a mechanism for key derivation with assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object

class **CKO_DOMAIN_PARAMETERS** and key type **CKK_GOSTR3410**. An algorithm for key derivation from [RFC 4357] (section 5.2) must be used.

The mechanism contributes the result as the **CKA_VALUE** attribute of the new private key. All other attributes must be specified in a template for creating private key object.

6.58 ChaCha20

ChaCha20 is a secret-key stream cipher described in [CHACHA].

Table 248, 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	✓					✓	

6.58.1 Definitions

This section defines the key type “CKK_CHACHA20” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

- CKM_CHACHA20_KEY_GEN
- CKM_CHACHA20

6.58.2 ChaCha20 secret key objects

ChaCha20 secret key objects (object class CKO_SECRET_KEY, key type CKK_CHACHA20) hold ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the common attributes defined for this object class:

Table 249, 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

The following is a sample template for creating a ChaCha20 secret key object:

```
CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_CHACHA20;
CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
}
```

```

13885         {CKA_VALUE, value, sizeof(value)}
13886     };
13887     CKA_CHECK_VALUE: The value of this attribute is derived from the key object by taking the first
13888     three bytes of the SHA-1 hash of the ChaCha20 secret key object's CKA_VALUE attribute.

```

13889 6.58.3 ChaCha20 mechanism parameters

13890 ♦ CK_CHACHA20_PARAMS; CK_CHACHA20_PARAMS_PTR

13891 **CK_CHACHA20_PARAMS** provides the parameters to the **CKM_CHACHA20** mechanism. It is defined
13892 as follows:

```

13893     typedef struct CK_CHACHA20_PARAMS {
13894         CK_BYTE_PTR    pBlockCounter;
13895         CK_ULONG        blockCounterBits;
13896         CK_BYTE_PTR    pNonce;
13897         CK_ULONG        ulNonceBits;
13898     } CK_CHACHA20_PARAMS;

```

13899 The fields of the structure have the following meanings:

13900	<i>pBlockCounter</i>	<i>pointer to block counter</i>
13901	<i>ulBlockCounterBits</i>	<i>length of block counter in bits (can be either 32 or 64)</i>
13902	<i>pNonce</i>	<i>nonce (This should be never re-used with the same key.)</i>
13903	<i>ulNonceBits</i>	<i>length of nonce in bits (is 64 for original, 96 for IETF and 192 for</i>
13904		<i>xchacha20 variant)</i>

13905 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)
13906 it is necessary to address these blocks in random order, thus this counter is exposed here.

13907 **CK_CHACHA20_PARAMS_PTR** is a pointer to **CK_CHACHA20_PARAMS**.

13908 6.58.4 ChaCha20 key generation

13909 The ChaCha20 key generation mechanism, denoted **CKM_CHACHA20_KEY_GEN**, is a key generation
13910 mechanism for ChaCha20.

13911 It does not have a parameter.

13912 The mechanism generates ChaCha20 keys of 256 bits.

13913 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
13914 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
13915 supports) may be specified in the template for the key, or else are assigned default initial values.

13916 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13917 specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is
13918 fixed at 256 bits.

13919

13920 6.58.5 ChaCha20 mechanism

13921 ChaCha20, denoted **CKM_CHACHA20**, is a mechanism for single and multiple-part encryption and
13922 decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and
13923 handling of their nonces, affecting the safety of using random nonces and the maximum size that can be
13924 encrypted safely.

13925 ChaCha20 has a parameter, **CK_CHACHA20_PARAMS**, which indicates the nonce and initial block
13926 counter value.
13927 Constraints on key types and the length of input and output data are summarized in the following table:
13928 *Table 250, 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

13929 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
13930 specify the supported range of ChaCha20 key sizes, in bits.
13931 *Table 251, 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.

13932 Nonces must not ever be reused with the same key. However due to the birthday paradox the first two
13933 variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended
13934 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.
13935 Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with
13936 randomly generated nonces the birthday paradox.

13937 **6.59 Salsa20**

13938 Salsa20 is a secret-key stream cipher described in **[SALSA]**.
13939 *Table 252, 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	✓					✓	

6.59.1 Definitions

This section defines the key type “CKK_SALSA20” and “CKK_SALSA20” for type CK_KEY_TYPE as used in the CKA_KEY_TYPE attribute of key objects.

Mechanisms:

CKM_SALSA20_KEY_GEN

CKM_SALSA20

6.59.2 Salsa20 secret key objects

Salsa20 secret key objects (object class CKO_SECRET_KEY, key type CKK_SALSA20) hold Salsa20 keys. The following table defines the Salsa20 secret key object attributes, in addition to the common attributes defined for this object class:

Table 253, 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

The following is a sample template for creating a Salsa20 secret key object:

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_SALSA20;
CK_UTF8CHAR label[] = "A Salsa20 secret key object";
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_ENCRYPT, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

```

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 ChaCha20 secret key object’s CKA_VALUE attribute.

6.59.3 Salsa20 mechanism parameters

◆ CK_SALSA20_PARAMS; CK_SALSA20_PARAMS_PTR

CK_SALSA20_PARAMS provides the parameters to the **CKM_SALSA20** mechanism. It is defined as follows:

```
typedef struct CK_SALSA20_PARAMS {  
    CK_BYTE_PTR    pBlockCounter;  
    CK_BYTE_PTR    pNonce;  
    CK_ULONG       ulNonceBits;  
} CK_SALSA20_PARAMS;
```

The fields of the structure have the following meanings:

<i>pBlockCounter</i>	<i>pointer to block counter (64 bits)</i>
<i>pNonce</i>	<i>nonce</i>
<i>ulNonceBits</i>	<i>size of the nonce in bits (64 for classic and 192 for XSalsa20)</i>

The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption) it is necessary to address these blocks in random order, thus this counter is exposed here.

CK_SALSA20_PARAMS_PTR is a pointer to **CK_SALSA20_PARAMS**.

6.59.4 Salsa20 key generation

The Salsa20 key generation mechanism, denoted **CKM_SALSA20_KEY_GEN**, is a key generation mechanism for Salsa20.

It does not have a parameter.

The mechanism generates Salsa20 keys of 256 bits.

The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the 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.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed at 256 bits.

6.59.5 Salsa20 mechanism

Salsa20, denoted **CKM_SALSA20**, is a mechanism for single and multiple-part encryption and decryption based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and handling of their nonces, affecting the safety of using random nonces.

Salsa20 has a parameter, **CK_SALSA20_PARAMS**, which indicates the nonce and initial block counter value.

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

Table 254, 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

14004 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14005 specify the supported range of ChaCha20 key sizes, in bits.

14006 *Table 255, 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.

14007 Nonces must not ever be reused with the same key. However due to the birthday paradox the original
14008 variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended
14009 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.
14010 Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly
14011 generated nonces the birthday paradox.

14012 **6.60 Poly1305**

14013 Poly1305 is a message authentication code designed by D.J Bernsterin **[POLY1305]**. Poly1305 takes a
14014 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

14015 *Table 256, 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		✓					

14016 **6.60.1 Definitions**

14017 This section defines the key type “CKK_POLY1305” for type CK_KEY_TYPE as used in the
14018 CKA_KEY_TYPE attribute of key objects.

14019 Mechanisms:

14020 CKM_POLY1305_KEY_GEN

14021 CKM_POLY1305

14022 **6.60.2 Poly1305 secret key objects**

14023 Poly1305 secret key objects (object class CKO_SECRET_KEY, key type CKK_POLY1305) hold
14024 Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the
14025 common attributes defined for this object class:

14026 *Table 257, 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

The following is a sample template for creating a Poly1305 secret key object:

```

CK_OBJECT_CLASS class = CKO_SECRET_KEY;
CK_KEY_TYPE keyType = CKK_POLY1305;
CK_UTF8CHAR label[] = "A Poly1305 secret key object";
CK_BYTE value[32] = {...};
CK_BBOOL true = CK_TRUE;
CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &class, sizeof(class)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SIGN, &true, sizeof(true)},
    {CKA_VALUE, value, sizeof(value)}
};

```

6.60.3 Poly1305 mechanism

Poly1305, denoted **CKM_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key and arbitrary length input.

It has no parameters.

Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

Table 258, Poly1305: Key and Data Length

Function	Key type	Data length	Signature Length
C_Sign	Poly1305	Any	128 bits
C_Verify	Poly1305	Any	128 bits

6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption

The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

Table 259, Poly1305 Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/Key Pair	Wrap & Unwrap	Derive
CKM_CHACHA20_POLY1305	✓						
CKM_SALSA20_POLY1305	✓						

6.61.1 Definitions

Mechanisms:

CKM_CHACHA20_POLY1305

CKM_SALSA20_POLY1305

6.61.2 Usage

Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305]. To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305 and Salsa20/Poly1305 both use CK_SALSA20_CHACHA20_POLY1305_PARAMS for Encrypt, Decrypt and CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS for MessageEncrypt, and MessageDecrypt.

Encrypt:

- Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data *pNonce* in the parameter block.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- Call C_EncryptInit() for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with parameters and key *K*.
- Call C_Encrypt(), or C_EncryptUpdate()*¹⁰ C_EncryptFinal(), for the plaintext obtaining ciphertext and authentication tag output.

Decrypt:

- Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data *pNonce* in the parameter block.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- Call C_DecryptInit() for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with parameters and key *K*.
- Call C_Decrypt(), or C_DecryptUpdate()*¹ C_DecryptFinal(), for the ciphertext, including the appended tag, obtaining plaintext output. Note: since **CKM_CHACHA20_POLY1305** and **CKM_SALSA20_POLY1305** are AEAD ciphers, no data should be returned until C_Decrypt() or C_DecryptFinal().

MessageEncrypt::

- Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- Set the Nonce data *pNonce* in the parameter block.
- Set *pTag* to hold the tag data returned from C_EncryptMessage() or the final C_EncryptMessageNext().
- Call C_MessageEncryptInit() for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305** mechanism with key *K*.

¹⁰ "*" indicates 0 or more calls may be made as required

- 14093 • Call C_EncryptMessage(), or C_EncryptMessageBegin followed by C_EncryptMessageNext()^{*11}.
14094 The mechanism parameter is passed to all three of these functions.
- 14095 • Call C_MessageEncryptFinal() to close the message encryption.

14096 MessageDecrypt:

- 14097 • Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of ChaCha20
14098 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 14099 • Set the Nonce data *pNonce* in the parameter block.
- 14100 • Set the tag data *pTag* in the parameter block before C_DecryptMessage or the final
14101 C_DecryptMessageNext()
- 14102 • Call C_MessageDecryptInit() for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
14103 mechanism with key *K*.
- 14104 • Call C_DecryptMessage(), or C_DecryptMessageBegin followed by C_DecryptMessageNext()^{*12}.
14105 The mechanism parameter is passed to all three of these functions.
- 14106 • Call C_MessageDecryptFinal() to close the message decryption

14107

14108 *ulNonceLen* is the length of the nonce in bits.

14109 In Encrypt and Decrypt the tag is appended to the cipher text. In MessageEncrypt the tag is returned in
14110 the *pTag* field of CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS. In MessageDecrypt the tag is
14111 provided by the *pTag* field of CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS. The application
14112 must provide 16 bytes of space for the tag.

14113 The key type for *K* must be compatible with **CKM_CHACHA20** or **CKM_SALSA20** respectively and the
14114 C_EncryptInit/C_DecryptInit calls shall behave, with respect to *K*, as if they were called directly with
14115 **CKM_CHACHA20** or **CKM_SALSA20**, *K* and NULL parameters.

14116 Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose
14117 the block counter, as the AEAD construction is based on a message metaphor in which random access is
14118 not needed.

14119 6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters

14120 ♦ **CK_SALSA20_CHACHA20_POLY1305_PARAMS;** 14121 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR**

14122 **CK_SALSA20_CHACHA20_POLY1305_PARAMS** is a structure that provides the parameters to the
14123 **CKM_CHACHA20_POLY1305** and **CKM_SALSA20_POLY1305** mechanisms. It is defined as follows:

```
14124 typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {
14125     CK_BYTE_PTR pNonce;
14126     CK_ULONG    ulNonceLen;
14127     CK_BYTE_PTR pAAD;
14128     CK_ULONG    ulAADLen;
14129 } CK_SALSA20_CHACHA20_POLY1305_PARAMS;
```

14130 The fields of the structure have the following meanings:

11 "*" indicates 0 or more calls may be made as required

12 "*" indicates 0 or more calls may be made as required

14131 *pNonce* *nonce (This should be never re-used with the same key.)*

14132 *ulNonceLen* *length of nonce in bits (is 64 for original, 96 for IETF (only for*
14133 *chacha20) and 192 for xchacha20/xsalsa20 variant)*

14134 *pAAD* *pointer to additional authentication data. This data is authenticated*
14135 *but not encrypted.*

14136 *ulAADLen* *length of pAAD in bytes.*

14137 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR** is a pointer to a
14138 **CK_SALSA20_CHACHA20_POLY1305_PARAMS**.

14139 ♦ **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;**
14140 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR**

14141 CK_CHACHA20POLY1305_PARAMS is a structure that provides the parameters to the CKM_
14142 CHACHA20_POLY1305 mechanism. It is defined as follows:

```
14143 typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {
14144     CK_BYTE_PTR    pNonce;
14145     CK_ULONG       ulNonceLen;
14146     CK_BYTE_PTR    pTag;
14147 } CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
```

14148 The fields of the structure have the following meanings:

14149 *pNonce* *pointer to nonce*

14150 *ulNonceLen* *length of nonce in bits. The length of the influences which variant of*
14151 *the ChaCha20 will be used (64 original, 96 IETF(only for*
14152 *ChaCha20), 192 XChaCha20/XSalsa20)*

14153 *pTag* *location of the authentication tag which is returned on*
14154 *MessageEncrypt, and provided on MessageDecrypt.*

14155 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR** is a pointer to a
14156 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS**.

14157 6.62 HKDF Mechanisms

14158 Details for HKDF key derivation mechanisms can be found in [RFC 5869].

14159

14160 Table 260, 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					✓		

6.62.1 Definitions

Mechanisms:

CKM_HKDF_DERIVE
CKM_HKDF_DATA
CKM_HKDF_KEY_GEN

Key Types:

CKK_HKDF

6.62.2 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:

```
typedef struct CK_HKDF_PARAMS {  
    CK_BBOOL bExtract;  
    CK_BBOOL bExpand;  
    CK_MECHANISM_TYPE prfHashMechanism;  
    CK_ULONG ulSaltType;  
    CK_BYTE_PTR pSalt;  
    CK_ULONG ulSaltLen;  
    CK_OBJECT_HANDLE hSaltKey;  
    CK_BYTE_PTR pInfo;  
    CK_ULONG ulInfoLen;  
} CK_HKDF_PARAMS;
```

The fields of the structure have the following meanings:

bExtract	execute the extract portion of HKDF.
bExpand	execute the expand portion of HKDF.
prfHashMechanism	base hash used for the HMAC in the underlying HKDF operation.
ulSaltType	specifies how the salt for the extract portion of the KDF is supplied. CKF_HKDF_SALT_NULL no salt is supplied. CKF_HKDF_SALT_DATA salt is supplied as a data in pSalt with length ulSaltLen. CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey.
pSalt	pointer to the salt.
ulSaltLen	length of the salt pointed to in pSalt.
hSaltKey	object handle to the salt key.
pInfo	info string for the expand stage.
ulInfoLen	length of the info string for the expand stage.

CK_HKDF_PARAMS_PTR is a pointer to a **CK_HKDF_PARAMS**.

6.62.3 HKDF derive

HKDF derivation implements the HKDF as specified in [RFC 5869]. The two booleans **bExtract** and **bExpand** control whether the extract section of the HKDF or the expand section of the HKDF is in use.

It has a parameter, a **CK_HKDF_PARAMS** structure, which allows for the passing of the salt and or the expansion info. The structure contains the bools *bExtract* and *bExpand* which control whether the extract or expand portions of the HKDF is to be used. This structure is defined in Section 6.62.2.

The input key must be of type **CKK_HKDF** or **CKK_GENERIC_SECRET** and the length must be the size of the underlying hash function specified in *prfHashMechanism*. The exception is a data object which has the same size as the underlying hash function, and which may be supplied as an input key. In this case **bExtract** should be true and non-null salt should be supplied.

Either *bExtract* or *bExpand* must be set to true. If they are both set to true, input key is first extracted then expanded. The salt is used in the extraction stage. If **bExtract** is set to true and no salt is given, a 'zero' salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified by the RFC. If **bExpand** is set to true, **CKA_VALUE_LEN** should be set to the desired key length. If it is false **CKA_VALUE_LEN** may be set to the length of the hash, but that is not necessary as the mechanism will supply this value. The salt should be ignored if *bExtract* is false. The *plInfo* should be ignored if *bExpand* is set to false.

The mechanism also contributes the **CKA_CLASS**, and **CKA_VALUE** attributes to the new key. Other attributes may be specified in the template, or else are assigned default values.

The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object class is **CKO_SECRET_KEY**. However, since these facts are all implicit in the mechanism, there is no need to specify any of them.

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.

6.62.4 HKDF Data

HKDF Data derive mechanism, denoted **CKM_HKDF_DATA**, is identical to HKDF Derive except the output is a **CKO_DATA** object whose value is the result to the derive operation. Some tokens may restrict what data may be successfully derived based on the *plInfo* portion of the **CK_HKDF_PARAMS**. Tokens may reject requests based on the *plInfo* values. Allowed *plInfo* values are specified in the profile document and applications could then query the appropriate profile before depending on the mechanism.

6.62.5 HKDF Key gen

HKDF key gen, denoted **CKM_HKDF_KEY_GEN** generates a new random HKDF key. **CKA_VALUE_LEN** must be set in the template.

6.63 NULL Mechanism

CKM_NULL is a mechanism used to implement the trivial pass-through function.

14247 Table 261, CKM_NULL Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_NULL	✓	✓	✓	✓		✓	✓
¹ SR = SignRecover, VR = VerifyRecover							

14248

14249 6.63.1 Definitions

14250 Mechanisms:

14251 CKM_NULL

14252 6.63.2 CKM_NULL mechanism parameters

14253 CKM_NULL does not have a parameter.

14254

14255 When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

14256 When used for signing, the input data is copied to the signature. When used for signature verification, it
14257 compares the input data and the signature, and returns CKR_OK (indicating that both are identical) or
14258 CKR_SIGNATURE_INVALID.

14259 When used for digesting data, the input data is copied to the message digest.

14260 When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be
14261 wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be
14262 created.

14263 When used for deriving a key, the derived key has the same value as the base key.

14264

14265 6.64 IKE Mechanisms

14266

14267 Table 262, IKE Mechanisms vs. Functions

Mechanism	Functions						
	Encrypt & Decrypt	Sign & Verify	SR & VR ¹	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive
CKM_IKE2_PRF_PLUS_DERIVE							✓
CKM_IKE_PRF_DERIVE							✓
CKM_IKE1_PRF_DERIVE							✓
CKM_IKE1_EXTENDED_DERIVE							✓

14268

14269 6.64.1 Definitions

14270 Mechanisms:

14271 CKM_IKE2_PRF_PLUS_DERIVE
 14272 CKM_IKE_PRF_DERIVE
 14273 CKM_IKE1_PRF_DERIVE
 14274 CKM_IKE1_EXTENDED_DERIVE
 14275

14276 6.64.2 IKE mechanism parameters

14277 ♦ CK_IKE2_PRF_PLUS_DERIVE_PARAMS; 14278 CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR

14279 CK_IKE2_PRF_PLUS_DERIVE_PARAMS is a structure that provides the parameters to the
 14280 CKM_IKE2_PRF_PLUS_DERIVE mechanism. It is defined as follows:

```
14281 typedef struct CK_IKE2_PRF_PLUS_DERIVE_PARAMS {
14282     CK_MECHANISM_TYPE prfMechanism;
14283     CK_BBOOL bHasSeedKey;
14284     CK_OBJECT_HANDLE hSeedKey;
14285     CK_BYTE_PTR pSeedData;
14286     CK_ULONG ulSeedDataLen;
14287 } CK_IKE2_PRF_PLUS_DERIVE_PARAMS;
```

14288
 14289 The fields of the structure have the following meanings:

14290	prfMechanism	underlying MAC mechanism used to generate the prf
14291	bHasSeedKey	hSeed key is present
14292	hSeedKey	optional seed from key
14293	pSeedData	optional seed from data
14294	ulSeedDataLen	length of optional seed data. If no seed data is present this value is
14295		0

14296 CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR is a pointer to a
 14297 CK_IKE2_PRF_PLUS_DERIVE_PARAMS.
 14298

14299 ♦ CK_IKE_PRF_DERIVE_PARAMS; CK_IKE_PRF_DERIVE_PARAMS_PTR

14300 CK_IKE_PRF_DERIVE_PARAMS is a structure that provides the parameters to the
 14301 CKM_IKE_PRF_DERIVE mechanism. It is defined as follows:

```
14302  

14303 typedef struct CK_IKE_PRF_DERIVE_PARAMS {
14304     CK_MECHANISM_TYPE prfMechanism;
14305     CK_BBOOL bDataAsKey;
14306     CK_BBOOL bRekey;
14307     CK_BYTE_PTR pNi;
14308     CK_ULONG ulNiLen;
14309     CK_BYTE_PTR pNr;
14310     CK_ULONG ulNrLen;
14311     CK_OBJECT_HANDLE hNewKey;
14312 } CK_IKE_PRF_DERIVE_PARAMS;
```

14313

14314 The fields of the structure have the following meanings:

14315	prfMechanism	underlying MAC mechanism used to generate the prf
14316	bDataAsKey	Ni Nr is used as the key for the prf rather than baseKey
14317	bRekey	rekey operation. hNewKey must be present
14318	pNi	Ni value
14319	ulNiLen	length of Ni
14320	pNr	Nr value
14321	ulNrLen	length of Nr
14322	hNewKey	New key value to drive the rekey.

14323 **CK_IKE_PRF_DERIVE_PARAMS_PTR** is a pointer to a **CK_IKE_PRF_DERIVE_PARAMS**.

14324

14325 **◆ CK_IKE1_PRF_DERIVE_PARAMS; CK_IKE1_PRF_DERIVE_PARAMS_PTR**

14326 **CK_IKE1_PRF_DERIVE_PARAMS** is a structure that provides the parameters to the
14327 **CKM_IKE1_PRF_DERIVE** mechanism. It is defined as follows:

```
14328     typedef struct CK_IKE1_PRF_DERIVE_PARAMS {  
14329         CK_MECHANISM_TYPE prfMechanism;  
14330         CK_BBOOL bHasPrevKey;  
14331         CK_OBJECT_HANDLE hKeygxy;  
14332         CK_OBJECT_HANDLE hPrevKey;  
14333         CK_BYTE_PTR pCKYi;  
14334         CK_ULONG ulCKYiLen;  
14335         CK_BYTE_PTR pCKYr;  
14336         CK_ULONG ulCKYrLen;  
14337         CK_BYTE keyNumber;  
14338     } CK_IKE1_PRF_DERIVE_PARAMS;
```

14339

14340 The fields of the structure have the following meanings:

14341	prfMechanism	underlying MAC mechanism used to generate the prf
14342	bHasPrevkey	hPrevKey is present
14343	hKeygxy	handle to the exchanged g ^{xy} key
14344	hPrevKey	handle to the previously derived key
14345	pCKYi	CKYi value
14346	ulCKYiLen	length of CKYi
14347	pCKYr	CKYr value
14348	ulCKYrLen	length of CKYr
14349	keyNumber	unique number for this key derivation

14350 **CK_IKE1_PRF_DERIVE_PARAMS_PTR** is a pointer to a **CK_IKE1_PRF_DERIVE_PARAMS**.

14351

◆ CK_IKE1_EXTENDED_DERIVE_PARAMS; CK_IKE1_EXTENDED_DERIVE_PARAMS_PTR

CK_IKE1_EXTENDED_DERIVE_PARAMS is a structure that provides the parameters to the **CKM_IKE1_EXTENDED_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_IKE1_EXTENDED_DERIVE_PARAMS {
    CK_MECHANISM_TYPE prfMechanism;
    CK_BBOOL bHasKeygxy;
    CK_OBJECT_HANDLE hKeygxy;
    CK_BYTE_PTR pExtraData;
    CK_ULONG ulExtraDataLen;
} CK_IKE1_EXTENDED_DERIVE_PARAMS;
```

The fields of the structure have the following meanings:

prfMechanism	underlying MAC mechanism used to generate the prf
bHasKeygxy	hKeygxy key is present
hKeygxy	optional key g ^{xy}
pExtraData	optional extra data
ulExtraDataLen	length of optional extra data. If no extra data is present this value is 0

CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR is a pointer to a **CK_IKE2_PRF_PLUS_DERIVE_PARAMS**.

6.64.3 IKE PRF DERIVE

The IKE PRF Derive mechanism denoted **CKM_IKE_PRF_DERIVE** is used in IPSEC both IKEv1 and IKEv2 to generate an initial key that is used to generate additional keys. It takes a **CK_IKE_PRF_DERIVE_PARAMS** as a mechanism parameter. *baseKey* is the base key passed into **C_DeriveKey**. *baseKey* must be of type **CKK_GENERIC_SECRET** if *bDataAsKey* is TRUE and the key type of the underlying prf if *bDataAsKey* is FALSE. *hNewKey* must be of type **CKK_GENERIC_SECRET**. Depending on the parameter settings, it generates keys with a **CKA_VALUE** of:

1. prf(pNi|pNr, baseKey); (bDataAsKey=TRUE, bRekey=FALSE)
2. prf(baseKey, pNi|pNr); (bDataAsKey=FALSE, bRekey=FALSE)
3. prf(baseKey, ValueOf(hNewKey)| pNi | pNr); (bDataAsKey=FALSE, bRekey=TRUE)

The resulting output key is always the length of the underlying prf. The combination of *bDataAsKey*=TRUE and *bRekey*=TRUE is not allowed. If both are set, **CKR_ARGUMENTS_BAD** is returned.

Case 1 is used in

- a. ikev2 (RFC 5996) *baseKey* is called g^{ir}, the output is called SKEYSEED
- b. ikev1 (RFC 2409) *baseKey* is called g^{ir}, the output is called SKEYID

Case 2 is used in ikev1 (RFC 2409) inkey is called pre-shared-key, output is called SKEYID

Case 3 is used in ikev2 (RFC 5996) rekey case, *baseKey* is SK_d, *hNewKey* is g^{ir} (new), the output is called SKEYSEED. The derived key will have a length of the length of the underlying prf. If

CKA_VALUE_LEN is specified, it must equal the underlying prf or **CKR_KEY_SIZE_RANGE** is returned. If **CKA_KEY_TYPE** is not specified in the template, it will be the underlying key type of the prf.

14396

14397 6.64.4 IKEv1 PRF DERIVE

14398 The IKEv1 PRF Derive mechanism denoted **CKM_IKE1_PRF_DERIVE** is used in IPSEC IKEv1 to
14399 generate various additional keys from the initial SKEYID. It takes a **CK_IKE1_PRF_DERIVE_PARAMS**
14400 as a mechanism parameter. SKEYID is the base key passed into **C_DeriveKey**.

14401

14402 This mechanism derives a key with **CKA_VALUE** set to either:

14403 $\text{prf}(\text{baseKey}, \text{ValueOf}(\text{hKeygxy}) \parallel \text{pCKYi} \parallel \text{pCKYr} \parallel \text{key_number})$

14404 or

14405 $\text{prf}(\text{baseKey}, \text{ValueOf}(\text{hPrevKey}) \parallel \text{ValueOf}(\text{hKeygxy}) \parallel \text{pCKYi} \parallel \text{pCKYr} \parallel \text{key_number})$

14406 depending on the state of *bHasPrevKey*.

14407 The key type of *baseKey* must be the key type of the prf, and the key type of *hKeygxy* must be
14408 **CKK_GENERIC_SECRET**. The key type of *hPrevKey* can be any key type.

14409

14410 This is defined in RFC 2409. For each of the following keys.

14411 *baseKey* is SKEYID, *hKeygxy* is g^{xy}

14412 for outKey = SKEYID_d, *bHasPrevKey* = false, key_number = 0

14413 for outKey = SKEYID_a, *hPrevKey* = SKEYID_d, key_number = 1

14414 for outKey = SKEYID_e, *hPrevKey* = SKEYID_a, key_number = 2

14415 If **CKA_VALUE_LEN** is not specified, the resulting key will be the length of the prf. If **CKA_VALUE_LEN**
14416 is greater than the prf, **CKR_KEY_SIZE_RANGE** is returned. If it is less the key is truncated taking the
14417 left most bytes. The value **CKA_KEY_TYPE** must be specified in the template or
14418 **CKR_TEMPLATE_INCOMPLETE** is returned.

14419

14420 6.64.5 IKEv2 PRF PLUS DERIVE

14421 The IKEv2 PRF PLUS Derive mechanism denoted **CKM_IKE2_PRF_PLUS_DERIVE** is used in IPSEC
14422 IKEv2 to derive various additional keys from the initial SKEYSEED. It takes a
14423 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS** as a mechanism parameter. SKEYSEED is the base key
14424 passed into **C_DeriveKey**. The key type of *baseKey* must be the key type of the underlying prf. This
14425 mechanism uses the base key and a feedback version of the prf to generate a single key with sufficient
14426 bytes to cover all additional keys. The application will then use **CKM_EXTRACT_KEY_FROM_KEY**
14427 several times to pull out the various keys. **CKA_VALUE_LEN** must be set in the template and its value
14428 must not be bigger than 255 times the size of the prf function output or **CKR_KEY_SIZE_RANGE** will be
14429 returned. If **CKA_KEY_TYPE** is not specified, the output key type will be **CKK_GENERIC_SECRET**.

14430

14431 This mechanism derives a key with a **CKA_VALUE** of (from RFC 5996):

14432

14433 $\text{prfplus} = T1 \parallel T2 \parallel T3 \parallel T4 \parallel \dots \parallel Tn$

14434 where:

14435 $T1 = \text{prf}(K, S \parallel 0x01)$

14436 $T2 = \text{prf}(K, T1 \parallel S \parallel 0x02)$

14437 $T3 = \text{prf}(K, T3 \parallel S \parallel 0x03)$

14438 $T4 = \text{prf}(K, T4 \parallel S \parallel 0x04)$

14439 .

14440 $T_n = \text{prf}(K, T_{(n-1)} \parallel n)$
 14441 $K = \text{baseKey}, S = \text{valueOf}(h\text{SeedKey}) \parallel p\text{SeedData}$
 14442

14443 **6.64.6 IKEv1 Extended Derive**

14444 The IKE Extended Derive mechanism denoted **CKM_IKE1_EXTENDED_DERIVE** is used in IPSEC
 14445 IKEv1 to derive longer keys than **CKM_IKE1_EXTENDED_DERIVE** can from the initial SKEYID. It is
 14446 used to support RFC 2409 appendix B and RFC 2409 section 5.5 (Quick Mode). It takes a
 14447 **CK_IKE1_EXTENDED_DERIVE_PARAMS** as a mechanism parameter. SKEYID is the base key passed
 14448 into **C_DeriveKey**. **CKA_VALUE_LEN** must be set in the template and its value must not be bigger than
 14449 255 times the size of the prf function output or **CKR_KEY_SIZE_RANGE** will be returned. If
 14450 **CKA_KEY_TYPE** is not specified, the output key type will be **CKK_GENERIC_SECRET**. The key type of
 14451 SKEYID must be the key type of the prf, and the key type of $h\text{Keygxy}$ (if present) must be
 14452 **CKK_GENERIC_SECRET**.

14453
 14454 This mechanism derives a key with **CKA_VALUE** (from RFC 2409 appendix B and section 5.5):
 14455 $K_a = K_1 \parallel K_2 \parallel K_3 \parallel K_4 \parallel \dots \parallel K_n$
 14456 where:
 14457 $K_1 = \text{prf}(K, \text{valueOf}(h\text{Keygxy}) \parallel p\text{ExtraData})$ or $\text{prf}(K, 0x00)$ if $b\text{HashKeygxy}$ is FALSE and $u\text{ExtraData}$
 14458 is 0
 14459 $K_2 = \text{prf}(K, K_1 \parallel \text{valueOf}(h\text{Keygxy}) \parallel p\text{ExtraData})$
 14460 $K_3 = \text{prf}(K, K_2 \parallel \text{valueOf}(h\text{Keygxy}) \parallel p\text{ExtraData})$
 14461 $K_4 = \text{prf}(K, K_3 \parallel \text{valueOf}(h\text{Keygxy}) \parallel p\text{ExtraData})$
 14462 .
 14463 $K_n = \text{prf}(K, K_{(n-1)} \parallel \text{valueOf}(h\text{Keygxy}) \parallel p\text{ExtraData})$
 14464 $K = \text{baseKey}$
 14465

14466 If **CKA_VALUE_LEN** is less then or equal to the prf length and $b\text{HasKeygxy}$ is CK_FALSE, then the new
 14467 key is simply the base key truncated to **CKA_VALUE_LEN** (specified in RFC 2409 appendix B).
 14468 Otherwise the prf is executed and the derived keys value is **CKA_VALUE_LEN** bytes of the resulting prf.

14469 **6.65 HSS**

14470 HSS is a mechanism for single-part signatures and verification, following the digital signature algorithm
 14471 defined in [RFC 8554] and [NIST 802-208].

14472
 14473 *Table 263, HSS Mechanisms vs. Functions*

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

14474 ¹ Single-part operations only
 14475

6.65.1 Definitions

This section defines the key type **CKK_HSS** for type **CK_KEY_TYPE** as used in the **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

Mechanisms:

CKM_HSS_KEY_PAIR_GEN

CKM_HSS

6.65.2 HSS public key objects

HSS public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_HSS**) hold HSS public keys.

The following table defines the HSS public key object attributes, in addition to the common attributes defined for this object class:

Table 264, HSS Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_HSS_LEVELS ^{2,4}	CK_ULONG	The number of levels in the HSS scheme.
CKA_HSS_LMS_TYPE ^{2,4}	CK_ULONG	The encoding for the Merkle tree heights of the top level LMS tree in the hierarchy.
CKA_HSS_LMOTS_TYPE ^{2,4}	CK_ULONG	The encoding for the Winternitz parameter of the one-time-signature scheme of the top level LMS tree.
CKA_VALUE ^{1,4}	Byte array	XDR-encoded public key as defined in [RFC8554].

Refer to Table 11 for footnotes

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

```
CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
CK_KEY_TYPE keyType = CKK_HSS;
CK_UTF8CHAR label[] = "An HSS public key object";
CK_BYTE value[] = {...};
CK_BBOOL true = CK_TRUE;
CK_BBOOL false = CK_FALSE;

CK_ATTRIBUTE template[] = {
    {CKA_CLASS, &keyClass, sizeof(keyClass)},
    {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
    {CKA_TOKEN, &>false, sizeof(false)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_VALUE, value, sizeof(value)},
    {CKA_VERIFY, &>true, sizeof(true)}
};
```

6.65.3 HSS private key objects

HSS private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_HSS**) hold HSS private keys.

The following table defines the HSS private key object attributes, in addition to the common attributes defined for this object class:

Table 265, HSS Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_HSS_LEVELS ^{1,3}	CK_ULONG	The number of levels in the HSS scheme.
CKA_HSS_LMS_TYPES ^{1,3}	CK_ULONG_PTR	A list of encodings for the Merkle tree heights of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK_ULONG. This number must match the CKA_HSS_LEVELS attribute value.
CKA_HSS_LMOTS_TYPES ^{1,3}	CK_ULONG_PTR	A list of encodings for the Winternitz parameter of the one-time-signature scheme of the LMS trees in the hierarchy from top to bottom. The number of encodings in the array is the ulValueLen component of the attribute divided by the size of CK_ULONG. This number must match the CKA_HSS_LEVELS attribute value.
CKA_VALUE ^{1,4,6,7}	Byte array	Vendor defined, must include state information. Note that exporting this value is dangerous as it would allow key reuse.
CKA_HSS_KEYS_REMAINING ^{2,4}	CK_ULONG	The minimum of the following two values: 1) The number of one-time private keys remaining; 2) $2^{32}-1$

Refer to Table 11 for footnotes

The encodings for CKA_HSS_LMOTS_TYPES and CKA_HSS_LMS_TYPES are defined in [RFC 8554] and [NIST 802-208].

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

```
CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_HSS;
CK_UTF8CHAR label[] = "An HSS private key object";
CK_ULONG hssLevels = 123;
CK_ULONG lmsTypes[] = {123, ...};
```

```

14524     CK_ULONG lmotsTypes[] = {123,...};
14525     CK_BYTE value[] = {...};
14526     CK_BBOOL true = CK_TRUE;
14527     CK_BBOOL false = CK_FALSE;
14528     CK_ATTRIBUTE template[] = {
14529         {CKA_CLASS, &keyClass, sizeof(keyClass)},
14530         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14531         {CKA_TOKEN, &true, sizeof(true)},
14532         {CKA_LABEL, label, sizeof(label)-1},
14533         {CKA_SENSITIVE, &true, sizeof(true)},
14534         {CKA_EXTRACTABLE, &false, sizeof(true)},
14535         {CKA_HSS_LEVELS, &hssLevels, sizeof(hssLevels)},
14536         {CKA_HSS_LMS_TYPES, lmsTypes, sizeof(lmsTypes)},
14537         {CKA_HSS_LMOTS_TYPES, lmotsTypes, sizeof(lmotsTypes)},
14538         {CKA_VALUE, value, sizeof(value)},
14539         {CKA_SIGN, &true, sizeof(true)}
14540     };

```

14541

14542 **CKA_SENSITIVE** MUST be true, **CKA_EXTRACTABLE** MUST be false, and **CKA_COPYABLE** MUST
14543 be false for this key.

14544 6.65.4 HSS key pair generation

14545 The HSS key pair generation mechanism, denoted **CKM_HSS_KEY_PAIR_GEN**, is a key pair generation
14546 mechanism for HSS.

14547 This mechanism does not have a parameter.

14548 The mechanism generates HSS public/private key pairs for the scheme specified by the
14549 **CKA_HSS_LEVELS**, **CKA_HSS_LMS_TYPES**, and **CKA_HSS_LMOTS_TYPES** attributes of the
14550 template for the private key.

14551 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_HSS_LEVELS**,
14552 **CKA_HSS_LMS_TYPE**, **CKA_HSS_LMOTS_TYPE**, and **CKA_VALUE** attributes to the new public key
14553 and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_HSS_KEYS_REMAINING** attributes
14554 to the new private key.

14555 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14556 are not used and must be set to 0.

14557 6.65.5 HSS without hashing

14558 The HSS without hashing mechanism, denoted **CKM_HSS**, is a mechanism for single-part signatures and
14559 verification for HSS. (This mechanism corresponds only to the part of LMS that processes the hash value,
14560 which may be of any length; it does not compute the hash value.)

14561 This mechanism does not have a parameter.

14562 For the purposes of these mechanisms, an HSS signature is a byte string with length depending on
14563 **CKA_HSS_LEVELS**, **CKA_HSS_LMS_TYPES**, **CKA_HSS_LMOTS_TYPES** as described in the
14564 following table.

14565 Table 266, HSS without hashing: Key and Data Length

Function	Key type	Input length	Output length
C_Sign ¹	HSS Private Key	any	1296-74988 ²
C_Verify ¹	HSS Public Key	any, 1296-74988 ²	N/A

14566 1 Single-part operations only.
14567 2 $4 + (levels - 1) * 56 + levels * (8 + (36 + 32 * p) + h * 32)$ where p has values (265, 133, 67, 34) for Imots type (W1, W2, W4, W8) and h is the number of levels in the LMS
14568 Merkle trees.

14569 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14570 are not used and must be set to 0.

14571 If the number of signatures is exhausted, CKR_KEY_EXHAUSTED will be returned.

14572

7 PKCS #11 Implementation Conformance

14573

7.1 PKCS#11 Consumer Implementation Conformance

14574

An implementation is a conforming PKCS#11 Consumer if the implementation meets the conditions specified in one or more consumer profiles specified in **[PKCS11-Prof]**.

14575

14576

A PKCS#11 consumer implementation SHALL be a conforming PKCS#11 Consumer.

14577

14578

14579

If a PKCS#11 consumer implementation claims support for a particular consumer profile, then the implementation SHALL conform to all normative statements within the clauses specified for that profile and for any subclauses to each of those clauses.

14580

7.2 PKCS#11 Provider Implementation Conformance

14581

14582

An implementation is a conforming PKCS#11 Provider if the implementation meets the conditions specified in one or more provider profiles specified in **[PKCS11-Prof]**.

14583

A PKCS#11 provider implementation SHALL be a conforming PKCS#11 Provider.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

Participants:

Salutation	First Name	Last Name	Company
Dr.	Warren	Armstrong	QuintessenceLabs Pty Ltd.
	Anthony	Berglas	Cryptsoft Pty Ltd.
Mr.	Dieter	Bong	Utimaco IS GmbH
Mr.	Roland	Bramm	PrimeKey Solutions AB
	Andrew	Byrne	Dell
	Hamish	Cameron	nCipher
	Kenli	Chong	QuintessenceLabs Pty Ltd.
Mr.	Justin	Corlett	Cryptsoft Pty Ltd.
	Xuele	Fan	Oracle
Mr.	Jan	Friedel	Oracle
Ms.	Susan	Gleeson	Oracle
Mr.	Thomas	Hardjono	M.I.T.
Mrs.	Jane	Harnad	OASIS
	David	Horton	Dell
	Tim	Hudson	Cryptsoft Pty Ltd.
Mr.	Gershon	Janssen	Individual
Mr.	Jakub	Jelen	Red Hat
Dr.	Mark	Joseph	P6R, Inc
Mr.	Paul	King	nCipher
Ms.	Dina	Kurktchi-Nimeh	Oracle
	John	Leiseboer	QuintessenceLabs Pty Ltd.
Mr.	John	Leser	Oracle
	Chris	Malafis	Red Hat
Dr.	Michael	Markowitz	Information Security Corporation
Mr.	Scott	Marshall	Cryptsoft Pty Ltd.

Salutation	First Name	Last Name	Company
Mr.	Chris	Meyer	Utimaco IS GmbH
Mr.	Darren	Moffat	Oracle
Dr.	Florian	Poppa	QuintessenceLabs Pty Ltd.
	Roland	Reichenberg	Utimaco IS GmbH
Mr.	Robert	Relyea	Red Hat
Mr.	Jonathan	Schulze-Hewett	Information Security Corporation
Mr.	Greg	Scott	Cryptsoft Pty Ltd.
Mr.	Martin	Shannon	QuintessenceLabs Pty Ltd.
Mr.	Oscar	So	Individual
	Patrick	Steuer	IBM
Mr.	Gerald	Stueve	Fornetix
	Jim	Susoy	P6R, Inc
Mr.	Sander	Temme	nCipher
Mr.	Manish	Upasani	Utimaco IS GmbH
Mr.	Charles	White	Fornetix
Ms.	Magda	Zdunkiewicz	Cryptsoft Pty Ltd.

14588

Appendix B. Manifest constants

The definitions for manifest constants specified in this document can be found in the following normative computer language definition files:

- `pkcs11.h` : <https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11.h>
- `pkcs11f.h` : <https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11f.h>
- `pkcs11t.h` : <https://github.com/oasis-tcs/pkcs11/blob/master/published/3-01/pkcs11t.h>

14596

Appendix C. Revision History

14597

Revision	Date	Editor	Changes Made
WD01	02 December 2020	Dieter Bong & Tony Cox	<ul style="list-style-type: none"> - Merged Base Specification & Current Mechanisms forming new “PKCS#11 Specification v3.1” - Added CKA_DERIVE_TEMPLATE
WD02	04 December 2020	Dieter Bong & Tony Cox	<ul style="list-style-type: none"> - Removed section 4.9.1 (covered in 6.1.3)
WD03	4 March 2021	Dieter Bong & Tony Cox	<ul style="list-style-type: none"> - Section 6.3.8 2nd paragraph replace “Edwards” by “Montgomery” - Revised Note in § 5.2
WD04	1 June 2021	Daniel Minder & Dieter Bong	<ul style="list-style-type: none"> - Fixed several references and typos - Moved CKM_SHA224_RSA_PKCS and CKM_SHA224_RSA_PKCS_PSS from table 137 to table 32 - Fixed the typo and added the wording wrt. CKA_VALUE_LEN in sections 6.64.2 and 6.64.6 - Section 4.9 Private key objects: replaced “<this version>” by “PKCS #11 V2.40” - Section 6.65 updated to HSS proposal dd. 12 May 2021 - Section 6.3: deprecation notice for CKM_ECDH_AES_KEY_WRAP
WD05	15 July 2021	Dieter Bong & Tony Cox	<ul style="list-style-type: none"> - Section 6.64: change the non-existing error CKR_KEY_RANGE_ERROR to CKR_KEY_SIZE_RANGE - Section 6.64.2: typo corrected: CK_IKE_PRF_PARAMS -> CK_IKE_PRF_DERIVE_PARAMS - Section 6.64.5: improved wording for IKE v2 key derivation - Section 6.64 and 6.65: formatting updated - Section 6.65: removed timeout error code and description - Section 5.9.5: Reported by Mostafa ADILI: C_EncryptMessageNext should be C_MessageEncryptFinal in the function declaration
WD06	14 October 2021	Dieter Bong & Tony Cox	<ul style="list-style-type: none"> - Added clarifying text to 6.64.6 - Clarified deprecation statement for CKM_ECDH_AES_KEY_WRAP - Updated [PKCS11-Prof] Reference - Clarified encodings in sections 6.3.5, 6.3.6, 6.3.7, 6.3.8, 6.3.16 & 6.3.17

Revision	Date	Editor	Changes Made
WD07	23 November 2021	Dieter Bong & Tony Cox	<ul style="list-style-type: none"> - Further clarification for CKM_ECDH_AES_KEY_WRAP deprecation notice - Clarified multiple EC key references in §6 (insertion of short Weierstrass descriptor) - Correction to description of CK_RSA_PKCS_MGF_TYPE
WD08	9 December 2021	Dieter Bong	<ul style="list-style-type: none"> - Clarified a few more EC key references in §6 (insertion of short Weierstrass descriptor and/or key type CKK_EC)
WD09	14 December 2021	Dieter Bong	<ul style="list-style-type: none"> - Updated deprecation notice for CKM_ECDH_AES_KEY_WRAP in section 6.3.20
WD10	21 January 2022	Dieter Bong	Removed deprecation notice for CKM_ECDH_AES_KEY_WRAP in section 6.3.20 as per TC meeting 12-January-2022
WD11	31 January 2022	Dieter Bong	Appendix B: include names of, and references to, computer language definition files

14598

Appendix D. Notices

Copyright © OASIS Open 2022. 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: [\[https://www.oasis-open.org/policies-guidelines/ipr/\]](https://www.oasis-open.org/policies-guidelines/ipr/).

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 AND ITS MEMBERS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THIS DOCUMENT OR ANY PART THEREOF.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specifications, OASIS Standards, or Approved Errata).

[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 Standards Final Deliverable, 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 deliverable.]

[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 OASIS Standards Final Deliverable 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 OASIS Standards Final Deliverable. 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 OASIS Standards Final Deliverable 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 Standards Final Deliverable, 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 document, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, documents, while reserving the right to enforce its marks against misleading uses. Please see <https://www.oasis-open.org/policies-guidelines/trademark/> for above guidance.