



PKCS #11 Specification Version 3.2

Committee Specification Draft 01

16 April 2025

This stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/pkcs11-spec-v3.2-csd01.docx> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/pkcs11-spec-v3.2-csd01.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/pkcs11-spec-v3.2-csd01.pdf>

Previous stage:

N/A

Latest stage:

<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/pkcs11-spec-v3.2.docx> (Authoritative)
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/pkcs11-spec-v3.2.html>
<https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/pkcs11-spec-v3.2.pdf>

Technical Committee:

OASIS PKCS 11 TC

Chairs:

Valerie Fenwick (vfenwick@apple.com), Apple, Inc.
Robert Relyea (rrelyea@redhat.com), Red Hat

Editors:

Dieter Bong (dieter.bong@utimaco.com), Utimaco IS GmbH
Greg Scott (greg.scott@cryptsoft.com), Cryptsoft Pty Ltd

Additional artifacts:

This prose specification is one component of a Work Product that also includes PKCS #11 header files:

- pkcs11.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/include/pkcs11-v3.2/pkcs11.h>
- pkcs11f.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/include/pkcs11-v3.2/pkcs11f.h>
- pkcs11t.h: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/include/pkcs11-v3.2/pkcs11t.h>

Related work:

This specification replaces or supersedes:

- *PKCS #11 Specification Version 3.1*. Edited by Dieter Bong and Tony Cox. OASIS Standard. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.1/pkcs11-spec-v3.1.html>.

This specification is related to:

- *PKCS #11 Profiles Version 3.2*. Edited by Tim Hudson. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.2/pkcs11-profiles-v3.2.html>.
- *PKCS #11 Usage Guide Version 3.2*. Edited by Dieter Bong. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-ug/v3.2/pkcs11-ug-v3.2.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 [[RFC 2119](#)] and [[RFC 8174](#)] 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-v3.2]

PKCS #11 Specification Version 3.2. Edited by Dieter Bong and Greg Scott. 16 April 2025. OASIS Committee Specification Draft 01. <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/csd01/pkcs11-spec-v3.2-csd01.html>. Latest stage: <https://docs.oasis-open.org/pkcs11/pkcs11-spec/v3.2/pkcs11-spec-v3.2.html>.

Notices:

Copyright © OASIS Open 2025. 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

77	1	Introduction.....	19
78	1.1	Definitions.....	19
79	1.2	Symbols and abbreviations.....	21
80	2	Platform- and compiler-dependent directives for C or C++	24
81	2.1	Structure packing.....	24
82	2.2	Pointer-related macros	24
83	3	General data types	26
84	3.1	General information	26
85	3.2	Slot and token types	27
86	3.3	Session types	32
87	3.4	Object types.....	34
88	3.5	Data types for mechanisms	38
89	3.6	Function types	40
90	3.7	Locking-related types.....	48
91	4	Objects	51
92	4.1	Creating, modifying, and copying objects.....	52
93	4.1.1	Creating objects	52
94	4.1.2	Modifying objects.....	53
95	4.1.3	Copying objects.....	53
96	4.2	Common attributes	54
97	4.3	Hardware Feature Objects.....	54
98	4.3.1	Definitions.....	54
99	4.3.2	Overview	54
100	4.3.3	Clock	55
101	4.3.3.1	Definition	55
102	4.3.3.2	Description	55
103	4.3.4	Monotonic Counter Objects.....	55
104	4.3.4.1	Definition	55
105	4.3.4.2	Description	55
106	4.3.5	User Interface Objects.....	55
107	4.3.5.1	Definition	55
108	4.3.5.2	Description	56
109	4.4	Storage Objects	56
110	4.4.1	The CKA_UNIQUE_ID attribute	57
111	4.5	Data objects.....	58
112	4.5.1	Definitions.....	58
113	4.5.2	Overview	58
114	4.6	Certificate objects	58
115	4.6.1	Definitions.....	58
116	4.6.2	Overview	58
117	4.6.3	X.509 public key certificate objects	59
118	4.6.4	WTLS public key certificate objects.....	61
119	4.6.5	X.509 attribute certificate objects	62
120	4.7	Trust objects	63

121	4.7.1 Definitions.....	63
122	4.7.2 Overview	64
123	4.8 Key objects	65
124	4.8.1 Definitions.....	65
125	4.8.2 Overview	66
126	4.9 Public key objects	67
127	4.10 Private key objects.....	68
128	4.11 Secret key objects	70
129	4.12 Domain parameter objects.....	72
130	4.12.1 Definitions.....	72
131	4.12.2 Overview	73
132	4.13 Mechanism objects	73
133	4.13.1 Definitions.....	73
134	4.13.2 Overview	73
135	4.14 Profile objects	73
136	4.14.1 Definitions.....	73
137	4.14.2 Overview	74
138	4.15 Validation objects.....	74
139	4.15.1 Definitions.....	74
140	4.15.2 Overview	74
141	4.15.3 Validation Indicators	75
142	4.15.3.1 Session validation flags.....	75
143	4.15.3.2 Key object state	75
144	5 Functions.....	77
145	5.1 Function return values	81
146	5.1.1 Universal Cryptoki function return values.....	81
147	5.1.2 Cryptoki function return values for functions that use a session handle	82
148	5.1.3 Cryptoki function return values for functions that use a token.....	82
149	5.1.4 Special return value for application-supplied callbacks.....	83
150	5.1.5 Special return values for mutex-handling functions.....	83
151	5.1.6 All other Cryptoki function return values.....	83
152	5.1.7 More on relative priorities of Cryptoki errors	88
153	5.1.8 Error code “gotchas”	88
154	5.2 Conventions for functions returning output in a variable-length buffer	89
155	5.3 Disclaimer concerning sample code	89
156	5.4 General-purpose functions	90
157	5.4.1 C_Initialize.....	90
158	5.4.2 C_Finalize	91
159	5.4.3 C_GetInfo	91
160	5.4.4 C_GetFunctionList.....	92
161	5.4.5 C_GetInterfaceList	93
162	5.4.6 C_GetInterface	94
163	5.5 Slot and token management functions	95
164	5.5.1 C_GetSlotList	96
165	5.5.2 C_GetSlotInfo.....	97
166	5.5.3 C_GetTokenInfo	97

167	5.5.4 C_WaitForSlotEvent	98
168	5.5.5 C_GetMechanismList	99
169	5.5.6 C_GetMechanismInfo	100
170	5.5.7 C_InitToken	101
171	5.5.8 C_InitPIN	102
172	5.5.9 C_SetPIN	103
173	5.6 Session management functions	104
174	5.6.1 C_OpenSession	104
175	5.6.2 C_CloseSession	105
176	5.6.3 C_CloseAllSessions	106
177	5.6.4 C_GetSessionInfo	107
178	5.6.5 C_SessionCancel	107
179	5.6.6 C_GetOperationState	109
180	5.6.7 C_SetOperationState	110
181	5.6.8 C_Login	112
182	5.6.9 C_LoginUser	113
183	5.6.10 C_Logout	114
184	5.6.11 C_GetSessionValidationFlags	115
185	5.7 Object management functions	115
186	5.7.1 C_CreateObject	115
187	5.7.2 C_CopyObject	117
188	5.7.3 C_DestroyObject	119
189	5.7.4 C_GetObjectSize	119
190	5.7.5 C_GetAttributeValue	120
191	5.7.6 C_SetAttributeValue	122
192	5.7.7 C_FindObjectsInit	123
193	5.7.8 C_FindObjects	124
194	5.7.9 C_FindObjectsFinal	124
195	5.8 Encryption functions	125
196	5.8.1 C_EncryptInit	125
197	5.8.2 C_Encrypt	125
198	5.8.3 C_EncryptUpdate	126
199	5.8.4 C_EncryptFinal	127
200	5.9 Message-based encryption functions	129
201	5.9.1 C_MessageEncryptInit	129
202	5.9.2 C_EncryptMessage	129
203	5.9.3 C_EncryptMessageBegin	130
204	5.9.4 C_EncryptMessageNext	131
205	5.9.5 C_MessageEncryptFinal	132
206	5.10 Decryption functions	133
207	5.10.1 C_DecryptInit	133
208	5.10.2 C_Decrypt	134
209	5.10.3 C_DecryptUpdate	135
210	5.10.4 C_DecryptFinal	135
211	5.11 Message-based decryption functions	137
212	5.11.1 C_MessageDecryptInit	137

213	5.11.2 C_DecryptMessage	138
214	5.11.3 C_DecryptMessageBegin	138
215	5.11.4 C_DecryptMessageNext	139
216	5.11.5 C_MessageDecryptFinal	140
217	5.12 Message digesting functions	140
218	5.12.1 C_DigestInit	140
219	5.12.2 C_Digest	141
220	5.12.3 C_DigestUpdate	141
221	5.12.4 C_DigestKey	142
222	5.12.5 C_DigestFinal	142
223	5.13 Signing and MACing functions	143
224	5.13.1 C_SignInit	143
225	5.13.2 C_Sign	144
226	5.13.3 C_SignUpdate	144
227	5.13.4 C_SignFinal	145
228	5.13.5 C_SignRecoverInit	146
229	5.13.6 C_SignRecover	146
230	5.14 Message-based signing and MACing functions	147
231	5.14.1 C_MessageSignInit	148
232	5.14.2 C_SignMessage	148
233	5.14.3 C_SignMessageBegin	149
234	5.14.4 C_SignMessageNext	149
235	5.14.5 C_MessageSignFinal	150
236	5.15 Functions for verifying signatures and MACs	150
237	5.15.1 C_VerifyInit	150
238	5.15.2 C_Verify	151
239	5.15.3 C_VerifyUpdate	152
240	5.15.4 C_VerifyFinal	152
241	5.15.5 C_VerifyRecoverInit	153
242	5.15.6 C_VerifyRecover	153
243	5.15.7 C_VerifySignatureInit	155
244	5.15.8 C_VerifySignature	155
245	5.15.9 C_VerifySignatureUpdate	156
246	5.15.10 C_VerifySignatureFinal	156
247	5.16 Message-based functions for verifying signatures and MACs	157
248	5.16.1 C_MessageVerifyInit	157
249	5.16.2 C_VerifyMessage	158
250	5.16.3 C_VerifyMessageBegin	158
251	5.16.4 C_VerifyMessageNext	159
252	5.16.5 C_MessageVerifyFinal	160
253	5.17 Dual-function cryptographic functions	160
254	5.17.1 C_DigestEncryptUpdate	160
255	5.17.2 C_DecryptDigestUpdate	163
256	5.17.3 C_SignEncryptUpdate	166
257	5.17.4 C_DecryptVerifyUpdate	168
258	5.18 Key management functions	171

259	5.18.1 C_GenerateKey.....	171
260	5.18.2 C_GenerateKeyPair	172
261	5.18.3 C_WrapKey	174
262	5.18.4 C_UnwrapKey	175
263	5.18.5 C_DeriveKey	177
264	5.18.6 C_WrapKeyAuthenticated	179
265	5.18.7 C_UnwrapKeyAuthenticated	181
266	5.18.8 C_EncapsulateKey.....	183
267	5.18.9 C_DecapsulateKey.....	185
268	5.19 Random number generation functions.....	186
269	5.19.1 C_SeedRandom.....	186
270	5.19.2 C_GenerateRandom	187
271	5.20 Parallel function management functions.....	187
272	5.20.1 C_GetFunctionStatus	187
273	5.20.2 C_CancelFunction	188
274	5.21 Asynchronous function management functions	188
275	5.21.1 C_AsyncComplete.....	188
276	5.21.2 C_AsyncGetID.....	190
277	5.21.3 C_AsyncJoin	190
278	5.22 Callback functions.....	191
279	5.22.1 Surrender callbacks.....	191
280	5.22.2 Vendor-defined callbacks	191
281	6 Mechanisms	192
282	6.1 RSA	192
283	6.1.1 Definitions.....	192
284	6.1.2 RSA public key objects.....	193
285	6.1.3 RSA private key objects	194
286	6.1.4 PKCS #1 RSA key pair generation.....	196
287	6.1.5 X9.31 RSA key pair generation	196
288	6.1.6 PKCS #1 v1.5 RSA	197
289	6.1.7 PKCS #1 RSA OAEP mechanism parameters.....	198
290	6.1.8 PKCS #1 RSA OAEP	199
291	6.1.9 PKCS #1 RSA PSS mechanism parameters	200
292	6.1.10 PKCS #1 RSA PSS	200
293	6.1.11 ISO/IEC 9796 RSA.....	201
294	6.1.12 X.509 (raw) RSA	201
295	6.1.13 ANSI X9.31 RSA	203
296	6.1.14 PKCS #1 v1.5 RSA signature with hashing.....	203
297	6.1.15 PKCS #1 RSA PSS signature with hashing	204
298	6.1.16 ANSI X9.31 RSA signature with SHA-1	205
299	6.1.17 TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA	206
300	6.1.18 TPM 1.1b and TPM 1.2 PKCS #1 RSA OAEP	206
301	6.1.19 RSA AES KEY WRAP.....	207
302	6.1.20 RSA AES KEY WRAP mechanism parameters	208
303	6.1.21 FIPS 186-4	208
304	6.2 DSA	208

305	6.2.1 Definitions.....	209
306	6.2.2 DSA public key objects.....	210
307	6.2.3 DSA Key Restrictions	211
308	6.2.4 DSA private key objects	211
309	6.2.5 DSA domain parameter objects	212
310	6.2.6 DSA key pair generation	213
311	6.2.7 DSA domain parameter generation.....	213
312	6.2.8 DSA probabilistic domain parameter generation.....	213
313	6.2.9 DSA Shawe-Taylor domain parameter generation.....	214
314	6.2.10 DSA base domain parameter generation	214
315	6.2.11 DSA without hashing	214
316	6.2.12 DSA with hashing	215
317	6.2.13 FIPS 186-4	215
318	6.3 Elliptic Curve.....	216
319	6.3.1 EC Signatures	218
320	6.3.2 Definitions.....	218
321	6.3.3 Short Weierstrass Elliptic Curve public key objects	219
322	6.3.4 Short Weierstrass Elliptic Curve private key objects	220
323	6.3.5 Edwards Elliptic Curve public key objects	222
324	6.3.6 Edwards Elliptic Curve private key objects.....	222
325	6.3.7 Montgomery Elliptic Curve public key objects	223
326	6.3.8 Montgomery Elliptic Curve private key objects.....	224
327	6.3.9 Elliptic Curve key pair generation.....	225
328	6.3.10 Edwards Elliptic Curve key pair generation.....	226
329	6.3.11 Montgomery Elliptic Curve key pair generation.....	226
330	6.3.12 ECDSA without hashing	227
331	6.3.13 ECDSA with hashing	227
332	6.3.14 EdDSA.....	228
333	6.3.15 XEdDSA	229
334	6.3.16 EC mechanism parameters.....	229
335	6.3.17 Elliptic Curve Diffie-Hellman key derivation.....	234
336	6.3.18 Elliptic Curve Diffie-Hellman with cofactor key derivation	235
337	6.3.19 Elliptic Curve Menezes-Qu-Vanstone key derivation	236
338	6.3.20 ECDH AES KEY WRAP	236
339	6.3.21 ECDH COFACTOR AES KEY WRAP	238
340	6.3.22 ECDH Montgomery AES KEY WRAP	239
341	6.3.23 ECDH AES KEY WRAP mechanism parameters	241
342	6.3.24 FIPS 186-4	241
343	6.4 Diffie-Hellman	241
344	6.4.1 Definitions.....	242
345	6.4.2 Diffie-Hellman public key objects	242
346	6.4.3 X9.42 Diffie-Hellman public key objects	243
347	6.4.4 Diffie-Hellman private key objects	244
348	6.4.5 X9.42 Diffie-Hellman private key objects.....	245
349	6.4.6 Diffie-Hellman domain parameter objects	246
350	6.4.7 X9.42 Diffie-Hellman domain parameters objects	246

351	6.4.8 PKCS #3 Diffie-Hellman key pair generation	247
352	6.4.9 PKCS #3 Diffie-Hellman domain parameter generation	247
353	6.4.10 PKCS #3 Diffie-Hellman key derivation	248
354	6.4.11 X9.42 Diffie-Hellman mechanism parameters	248
355	6.4.12 X9.42 Diffie-Hellman key pair generation	251
356	6.4.13 X9.42 Diffie-Hellman domain parameter generation	251
357	6.4.14 X9.42 Diffie-Hellman key derivation	252
358	6.4.15 X9.42 Diffie-Hellman hybrid key derivation	252
359	6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation	253
360	6.5 Extended Triple Diffie-Hellman (x3dh)	254
361	6.5.1 Definitions	254
362	6.5.2 Extended Triple Diffie-Hellman key objects	254
363	6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange	254
364	6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange	255
365	6.5.5 Extended Triple Diffie-Hellman parameters	256
366	6.6 Double Ratchet	256
367	6.6.1 Definitions	257
368	6.6.2 Double Ratchet secret key objects	257
369	6.6.3 Double Ratchet key derivation	258
370	6.6.4 Double Ratchet Encryption mechanism	259
371	6.6.5 Double Ratchet parameters	259
372	6.7 Wrapping/unwrapping private keys	260
373	6.8 Generic secret key	264
374	6.8.1 Definitions	264
375	6.8.2 Generic secret key objects	264
376	6.8.3 Generic secret key generation	265
377	6.9 HMAC mechanisms	265
378	6.9.1 General block cipher mechanism parameters	265
379	6.10 AES	265
380	6.10.1 Definitions	266
381	6.10.2 AES secret key objects	266
382	6.10.3 AES key generation	267
383	6.10.4 AES-ECB	267
384	6.10.5 AES-CBC	268
385	6.10.6 AES-CBC with PKCS padding	269
386	6.10.7 AES-OFB	269
387	6.10.8 AES-CFB	270
388	6.10.9 General-length AES-MAC	270
389	6.10.10 AES-MAC	270
390	6.10.11 AES-XCBC-MAC	271
391	6.10.12 AES-XCBC-MAC-96	271
392	6.11 AES with Counter	271
393	6.11.1 Definitions	271
394	6.11.2 AES with Counter mechanism parameters	272
395	6.11.3 AES with Counter Encryption / Decryption	272
396	6.12 AES CBC with CipherText Stealing CTS	272

397	6.12.1 Definitions.....	273
398	6.12.2 AES CTS mechanism parameters	273
399	6.13 Additional AES Mechanisms.....	273
400	6.13.1 Definitions.....	273
401	6.13.2 AES-GCM Authenticated Encryption / Decryption	274
402	6.13.3 AES-GCM Authenticated Wrap / Unwrap.....	276
403	6.13.4 AES-CCM Authenticated Encryption / Decryption.....	277
404	6.13.5 AES-CCM Authenticated Wrap / Unwrap	279
405	6.13.6 AES-GMAC	281
406	6.13.7 AES GCM and CCM Mechanism parameters	281
407	6.14 AES CMAC	285
408	6.14.1 Definitions.....	286
409	6.14.2 Mechanism parameters.....	286
410	6.14.3 General-length AES-CMAC.....	286
411	6.14.4 AES-CMAC	286
412	6.15 AES XTS.....	287
413	6.15.1 Definitions.....	287
414	6.15.2 AES-XTS secret key objects	287
415	6.15.3 AES-XTS key generation	287
416	6.15.4 AES-XTS	287
417	6.16 AES Key Wrap.....	288
418	6.16.1 Definitions.....	288
419	6.16.2 AES Key Wrap Mechanism parameters.....	288
420	6.16.3 AES Key Wrap	288
421	6.17 Key derivation by data encryption – DES & AES.....	289
422	6.17.1 Definitions.....	289
423	6.17.2 Mechanism Parameters	290
424	6.17.3 Mechanism Description	290
425	6.18 Double and Triple-length DES	291
426	6.18.1 Definitions.....	291
427	6.18.2 DES2 secret key objects	291
428	6.18.3 DES3 secret key objects	292
429	6.18.4 Double-length DES key generation	293
430	6.18.5 Triple-length DES Order of Operations	293
431	6.18.6 Triple-length DES in CBC Mode.....	293
432	6.18.7 DES and Triple length DES in OFB Mode.....	293
433	6.18.8 DES and Triple length DES in CFB Mode.....	294
434	6.19 Double and Triple-length DES CMAC	294
435	6.19.1 Definitions.....	294
436	6.19.2 Mechanism parameters.....	295
437	6.19.3 General-length DES3-MAC	295
438	6.19.4 DES3-CMAC	295
439	6.20 SHA-1	295
440	6.20.1 Definitions.....	296
441	6.20.2 SHA-1 digest	296
442	6.20.3 General-length SHA-1-HMAC	296

443	6.20.4 SHA-1-HMAC	297
444	6.20.5 SHA-1 key derivation.....	297
445	6.20.6 SHA-1 HMAC key generation.....	297
446	6.21 SHA-224	298
447	6.21.1 Definitions.....	298
448	6.21.2 SHA-224 digest	298
449	6.21.3 General-length SHA-224-HMAC	298
450	6.21.4 SHA-224-HMAC	299
451	6.21.5 SHA-224 key derivation.....	299
452	6.21.6 SHA-224 HMAC key generation.....	299
453	6.22 SHA-256	299
454	6.22.1 Definitions.....	300
455	6.22.2 SHA-256 digest	300
456	6.22.3 General-length SHA-256-HMAC	300
457	6.22.4 SHA-256-HMAC	301
458	6.22.5 SHA-256 key derivation.....	301
459	6.22.6 SHA-256 HMAC key generation.....	301
460	6.23 SHA-384	301
461	6.23.1 Definitions.....	301
462	6.23.2 SHA-384 digest	302
463	6.23.3 General-length SHA-384-HMAC	302
464	6.23.4 SHA-384-HMAC	302
465	6.23.5 SHA-384 key derivation.....	302
466	6.23.6 SHA-384 HMAC key generation.....	303
467	6.24 SHA-512	303
468	6.24.1 Definitions.....	303
469	6.24.2 SHA-512 digest	303
470	6.24.3 General-length SHA-512-HMAC	304
471	6.24.4 SHA-512-HMAC	304
472	6.24.5 SHA-512 key derivation.....	304
473	6.24.6 SHA-512 HMAC key generation.....	304
474	6.25 SHA-512/224	305
475	6.25.1 Definitions.....	305
476	6.25.2 SHA-512/224 digest	305
477	6.25.3 General-length SHA-512/224-HMAC	305
478	6.25.4 SHA-512/224-HMAC	306
479	6.25.5 SHA-512/224 key derivation.....	306
480	6.25.6 SHA-512/224 HMAC key generation.....	306
481	6.26 SHA-512/256	306
482	6.26.1 Definitions.....	307
483	6.26.2 SHA-512/256 digest	307
484	6.26.3 General-length SHA-512/256-HMAC	307
485	6.26.4 SHA-512/256-HMAC	307
486	6.26.5 SHA-512/256 key derivation.....	308
487	6.26.6 SHA-512/256 HMAC key generation.....	308
488	6.27 SHA-512/t	308

489	6.27.1 Definitions.....	308
490	6.27.2 SHA-512/t digest	308
491	6.27.3 General-length SHA-512/t-HMAC	309
492	6.27.4 SHA-512/t-HMAC	309
493	6.27.5 SHA-512/t key derivation.....	309
494	6.27.6 SHA-512/t HMAC key generation.....	309
495	6.28 SHA3-224	309
496	6.28.1 Definitions.....	310
497	6.28.2 SHA3-224 digest	310
498	6.28.3 General-length SHA3-224-HMAC	310
499	6.28.4 SHA3-224-HMAC	311
500	6.28.5 SHA3-224 key derivation.....	311
501	6.28.6 SHA3-224 HMAC key generation.....	311
502	6.29 SHA3-256	311
503	6.29.1 Definitions.....	311
504	6.29.2 SHA3-256 digest	312
505	6.29.3 General-length SHA3-256-HMAC	312
506	6.29.4 SHA3-256-HMAC	312
507	6.29.5 SHA3-256 key derivation.....	312
508	6.29.6 SHA3-256 HMAC key generation.....	313
509	6.30 SHA3-384	313
510	6.30.1 Definitions.....	313
511	6.30.2 SHA3-384 digest	313
512	6.30.3 General-length SHA3-384-HMAC	314
513	6.30.4 SHA3-384-HMAC	314
514	6.30.5 SHA3-384 key derivation.....	314
515	6.30.6 SHA3-384 HMAC key generation.....	314
516	6.31 SHA3-512	315
517	6.31.1 Definitions.....	315
518	6.31.2 SHA3-512 digest	315
519	6.31.3 General-length SHA3-512-HMAC	315
520	6.31.4 SHA3-512-HMAC	316
521	6.31.5 SHA3-512 key derivation.....	316
522	6.31.6 SHA3-512 HMAC key generation.....	316
523	6.32 SHAKE.....	316
524	6.32.1 Definitions.....	316
525	6.32.2 SHAKE Key Derivation.....	317
526	6.33 BLAKE2B-160.....	317
527	6.33.1 Definitions.....	317
528	6.33.2 BLAKE2B-160 digest.....	318
529	6.33.3 General-length BLAKE2B-160-HMAC.....	318
530	6.33.4 BLAKE2B-160-HMAC	318
531	6.33.5 BLAKE2B-160 key derivation	318
532	6.33.6 BLAKE2B-160 HMAC key generation	318
533	6.34 BLAKE2B-256.....	319
534	6.34.1 Definitions.....	319

535	6.34.2 BLAKE2B-256 digest.....	319
536	6.34.3 General-length BLAKE2B-256-HMAC.....	319
537	6.34.4 BLAKE2B-256-HMAC	320
538	6.34.5 BLAKE2B-256 key derivation	320
539	6.34.6 BLAKE2B-256 HMAC key generation	320
540	6.35 BLAKE2B-384.....	320
541	6.35.1 Definitions.....	320
542	6.35.2 BLAKE2B-384 digest.....	321
543	6.35.3 General-length BLAKE2B-384-HMAC.....	321
544	6.35.4 BLAKE2B-384-HMAC	321
545	6.35.5 BLAKE2B-384 key derivation	321
546	6.35.6 BLAKE2B-384 HMAC key generation	322
547	6.36 BLAKE2B-512.....	322
548	6.36.1 Definitions.....	322
549	6.36.2 BLAKE2B-512 digest.....	322
550	6.36.3 General-length BLAKE2B-512-HMAC.....	323
551	6.36.4 BLAKE2B-512-HMAC	323
552	6.36.5 BLAKE2B-512 key derivation	323
553	6.36.6 BLAKE2B-512 HMAC key generation	323
554	6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE).....	324
555	6.37.1 Definitions.....	324
556	6.37.2 Password-based encryption/authentication mechanism parameters	324
557	6.37.3 PKCS #5 PBKDF2 key generation mechanism parameters	325
558	6.37.4 PKCS #5 PBKD2 key generation	327
559	6.38 PKCS #12 password-based encryption/authentication mechanisms	327
560	6.38.1 SHA-1-PBE for 3-key triple-DES-CBC	328
561	6.38.2 SHA-1-PBE for 2-key triple-DES-CBC	328
562	6.38.3 SHA-1-PBA for SHA-1-HMAC.....	328
563	6.39 SSL	328
564	6.39.1 Definitions.....	329
565	6.39.2 SSL mechanism parameters	329
566	6.39.3 Pre-master key generation	331
567	6.39.4 Master key derivation	331
568	6.39.5 Master key derivation for Diffie-Hellman	332
569	6.39.6 Key and MAC derivation.....	333
570	6.39.7 MD5 MACing in SSL 3.0	334
571	6.39.8 SHA-1 MACing in SSL 3.0	334
572	6.40 TLS 1.2 Mechanisms.....	334
573	6.40.1 Definitions.....	335
574	6.40.2 TLS 1.2 mechanism parameters	335
575	6.40.3 TLS MAC.....	339
576	6.40.4 Master key derivation	339
577	6.40.5 Master key derivation for Diffie-Hellman	340
578	6.40.6 Key and MAC derivation.....	341
579	6.40.7 CKM_TLS12_KEY_SAFE_DERIVE.....	342
580	6.40.8 Generic Key Derivation using the TLS PRF	342

581	6.40.9 Deprecated TLS 1.2 mechanisms	343
582	6.41 WTLS.....	343
583	6.41.1 Definitions.....	343
584	6.41.2 WTLS mechanism parameters.....	343
585	6.41.3 Pre master secret key generation for RSA key exchange suite	346
586	6.41.4 Master secret key derivation	346
587	6.41.5 Master secret key derivation for Diffie-Hellman and Elliptic Curve Cryptography	347
588	6.41.6 WTLS PRF (pseudorandom function)	348
589	6.41.7 Server Key and MAC derivation	348
590	6.41.8 Client key and MAC derivation	349
591	6.42 SP800-108 Key Derivation	350
592	6.42.1 Definitions.....	350
593	6.42.2 Mechanism Parameters	351
594	6.42.3 Counter Mode KDF	356
595	6.42.4 Feedback Mode KDF	357
596	6.42.5 Double Pipeline Mode KDF	358
597	6.42.6 Deriving Additional Keys	359
598	6.42.7 Key Derivation Attribute Rules	360
599	6.42.8 Constructing PRF Input Data	360
600	6.42.8.1 Sample Counter Mode KDF	360
601	6.42.8.2 Sample SCP03 Counter Mode KDF	361
602	6.42.8.3 Sample Feedback Mode KDF	362
603	6.42.8.4 Sample Double-Pipeline Mode KDF.....	363
604	6.43 Miscellaneous simple key derivation mechanisms	364
605	6.43.1 Definitions.....	365
606	6.43.2 Parameters for miscellaneous simple key derivation mechanisms	365
607	6.43.3 Concatenation of a base key and another key	366
608	6.43.4 Concatenation of a base key and data.....	366
609	6.43.5 Concatenation of data and a base key.....	367
610	6.43.6 XORing of a key and data	368
611	6.43.7 Extraction of one key from another key.....	368
612	6.43.8 Public key from private key	369
613	6.44 CMS.....	370
614	6.44.1 Definitions.....	370
615	6.44.2 CMS Signature Mechanism Objects	370
616	6.44.3 CMS mechanism parameters.....	371
617	6.44.4 CMS signatures.....	372
618	6.45 Blowfish	373
619	6.45.1 Definitions.....	373
620	6.45.2 BLOWFISH secret key objects.....	373
621	6.45.3 Blowfish key generation	374
622	6.45.4 Blowfish-CBC	374
623	6.45.5 Blowfish-CBC with PKCS padding	375
624	6.46 Twofish	375
625	6.46.1 Definitions.....	375
626	6.46.2 Twofish secret key objects	376
627	6.46.3 Twofish key generation	376

628	6.46.4 Twofish -CBC	376
629	6.46.5 Twofish-CBC with PKCS padding	377
630	6.47 CAMELLIA	377
631	6.47.1 Definitions.....	377
632	6.47.2 Camellia secret key objects.....	377
633	6.47.3 Camellia key generation.....	378
634	6.47.4 Camellia-ECB.....	378
635	6.47.5 Camellia-CBC.....	379
636	6.47.6 Camellia-CBC with PKCS padding.....	379
637	6.47.7 CAMELLIA with Counter mechanism parameters.....	380
638	6.47.8 General-length Camellia-MAC	381
639	6.47.9 Camellia-MAC	381
640	6.48 Key derivation by data encryption - Camellia	382
641	6.48.1 Definitions.....	382
642	6.48.2 Mechanism Parameters	382
643	6.49 ARIA	382
644	6.49.1 Definitions.....	383
645	6.49.2 Aria secret key objects	383
646	6.49.3 ARIA key generation	384
647	6.49.4 ARIA-ECB	384
648	6.49.5 ARIA-CBC	384
649	6.49.6 ARIA-CBC with PKCS padding	385
650	6.49.7 General-length ARIA-MAC	386
651	6.49.8 ARIA-MAC	386
652	6.50 Key derivation by data encryption - ARIA.....	386
653	6.50.1 Definitions.....	387
654	6.50.2 Mechanism Parameters	387
655	6.51 SEED	387
656	6.51.1 Definitions.....	388
657	6.51.2 SEED secret key objects.....	388
658	6.51.3 SEED key generation	389
659	6.51.4 SEED-ECB	389
660	6.51.5 SEED-CBC.....	389
661	6.51.6 SEED-CBC with PKCS padding.....	389
662	6.51.7 General-length SEED-MAC.....	389
663	6.51.8 SEED-MAC	389
664	6.52 Key derivation by data encryption - SEED.....	390
665	6.52.1 Definitions.....	390
666	6.52.2 Mechanism Parameters	390
667	6.53 OTP	390
668	6.53.1 Usage overview.....	390
669	6.53.2 Case 1: Generation of OTP values	391
670	6.53.3 Case 2: Verification of provided OTP values.....	392
671	6.53.4 Case 3: Generation of OTP keys	392
672	6.53.5 OTP objects.....	393
673	6.53.5.1 Key objects.....	393

674	6.53.6 OTP-related notifications.....	394
675	6.53.7 OTP mechanisms.....	394
676	6.53.7.1 OTP mechanism parameters	395
677	6.53.8 RSA SecurID	399
678	6.53.8.1 RSA SecurID secret key objects	399
679	6.53.8.2 RSA SecurID key generation	400
680	6.53.8.3 SecurID OTP generation and validation	400
681	6.53.8.4 Return values	400
682	6.53.9 OATH HOTP	400
683	6.53.9.1 OATH HOTP secret key objects.....	400
684	6.53.9.2 HOTP key generation	401
685	6.53.9.3 HOTP OTP generation and validation	401
686	6.53.10 ActivIdentity ACTI.....	402
687	6.53.10.1 ACTI secret key objects	402
688	6.53.10.2 ACTI key generation.....	403
689	6.53.10.3 ACTI OTP generation and validation.....	403
690	6.54 CT-KIP	404
691	6.54.1 Principles of Operation	404
692	6.54.2 Mechanisms	404
693	6.54.3 Definitions.....	405
694	6.54.4 CT-KIP Mechanism parameters	405
695	6.54.5 CT-KIP key derivation	405
696	6.54.6 CT-KIP key wrap and key unwrap.....	406
697	6.54.7 CT-KIP signature generation.....	406
698	6.55 GOST 28147-89	406
699	6.55.1 Definitions.....	406
700	6.55.2 GOST 28147-89 secret key objects	407
701	6.55.3 GOST 28147-89 domain parameter objects	407
702	6.55.4 GOST 28147-89 key generation	408
703	6.55.5 GOST 28147-89-ECB	409
704	6.55.6 GOST 28147-89 encryption mode except ECB	409
705	6.55.7 GOST 28147-89-MAC.....	410
706	6.55.8 GOST 28147-89 keys wrapping/unwrapping with GOST 28147-89.....	410
707	6.56 GOST R 34.11-94.....	411
708	6.56.1 Definitions.....	411
709	6.56.2 GOST R 34.11-94 domain parameter objects.....	411
710	6.56.3 GOST R 34.11-94 digest.....	412
711	6.56.4 GOST R 34.11-94 HMAC.....	413
712	6.57 GOST R 34.10-2001.....	413
713	6.57.1 Definitions.....	414
714	6.57.2 GOST R 34.10-2001 public key objects	414
715	6.57.3 GOST R 34.10-2001 private key objects.....	415
716	6.57.4 GOST R 34.10-2001 domain parameter objects.....	417
717	6.57.5 GOST R 34.10-2001 mechanism parameters.....	418
718	6.57.6 GOST R 34.10-2001 key pair generation.....	419
719	6.57.7 GOST R 34.10-2001 without hashing	420
720	6.57.8 GOST R 34.10-2001 with GOST R 34.11-94.....	420

721	6.57.9 GOST 28147-89 keys wrapping/unwrapping with GOST R 34.10-2001	420
722	6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys	421
723	6.58 ChaCha20.....	421
724	6.58.1 Definitions.....	421
725	6.58.2 ChaCha20 secret key objects	421
726	6.58.3 ChaCha20 mechanism parameters.....	422
727	6.58.4 ChaCha20 key generation.....	422
728	6.58.5 ChaCha20 mechanism.....	423
729	6.59 Salsa20.....	423
730	6.59.1 Definitions.....	424
731	6.59.2 Salsa20 secret key objects.....	424
732	6.59.3 Salsa20 mechanism parameters	424
733	6.59.4 Salsa20 key generation.....	425
734	6.59.5 Salsa20 mechanism	425
735	6.60 Poly1305.....	426
736	6.60.1 Definitions.....	426
737	6.60.2 Poly1305 secret key objects.....	426
738	6.60.3 Poly1305 mechanism	427
739	6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated Encryption / Decryption	427
740	6.61.1 Definitions.....	427
741	6.61.2 Usage	427
742	6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters.....	429
743	6.62 HKDF Mechanisms.....	430
744	6.62.1 Definitions.....	430
745	6.62.2 HKDF mechanism parameters	430
746	6.62.3 HKDF derive.....	431
747	6.62.4 HKDF Data	432
748	6.62.5 HKDF Key gen	432
749	6.63 NULL Mechanism	432
750	6.63.1 Definitions.....	432
751	6.63.2 CKM_NULL mechanism parameters.....	432
752	6.64 IKE Mechanisms.....	433
753	6.64.1 Definitions.....	433
754	6.64.2 IKE mechanism parameters	433
755	6.64.3 IKE PRF DERIVE	436
756	6.64.4 IKEv1 PRF DERIVE	436
757	6.64.5 IKEv2 PRF PLUS DERIVE	437
758	6.64.6 IKEv1 Extended Derive	437
759	6.65 HSS	438
760	6.65.1 Definitions.....	438
761	6.65.2 HSS public key objects.....	438
762	6.65.3 HSS private key objects	439
763	6.65.4 HSS key pair generation	441
764	6.65.5 HSS without hashing	441
765	6.66 XMSS and XMSS ^{MT}	442
766	6.66.1 Definitions.....	442

767	6.66.2 XMSS public key objects	442
768	6.66.3 XMSS ^{MT} public key objects	443
769	6.66.4 XMSS private key objects	444
770	6.66.5 XMSS ^{MT} private key objects	445
771	6.66.6 XMSS key pair generation.....	446
772	6.66.7 XMSS ^{MT} key pair generation	446
773	6.66.8 XMSS and XMSS ^{MT} without hashing.....	446
774	6.67 ML-DSA	447
775	6.67.1 Definitions.....	448
776	6.67.2 ML-DSA public key objects	449
777	6.67.3 ML-DSA private key objects	450
778	6.67.4 ML-DSA key pair generation	451
779	6.67.5 ML-DSA Signature.....	451
780	6.67.6 HashML-DSA Signature	451
781	6.67.7 HashML-DSA Signature with hashing	452
782	6.68 ML-KEM.....	453
783	6.68.1 Definitions.....	453
784	6.68.2 ML-KEM public key objects	453
785	6.68.3 ML-KEM private key objects.....	454
786	6.68.4 ML-KEM key pair generation.....	455
787	6.68.5 ML-KEM Key Agreement.....	455
788	6.69 SLH-DSA	456
789	6.69.1 Definitions.....	456
790	6.69.2 SLH-DSA public key objects	457
791	6.69.3 SLH-DSA private key objects	458
792	6.69.4 SLH-DSA key pair generation	459
793	6.69.5 SLH-DSA Signature.....	459
794	6.69.6 HashSLH-DSA Signature	460
795	6.69.7 HashSLH-DSA Signature with hashing	460
796	7 PKCS #11 Implementation Conformance	462
797	7.1 PKCS #11 Consumer Implementation Conformance	462
798	7.2 PKCS #11 Provider Implementation Conformance	462
799	Appendix A. References	463
800	A.1 Normative References.....	463
801	A.2 Informative References	466
802	Appendix B. Acknowledgments	471
803	B.1 Special Thanks.....	471
804	B.2 Participants.....	471
805	Appendix C. Revision History	472
806	Appendix D. Notices	475
807		

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.

This document also specifies details of cryptographic mechanisms (algorithms).

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.

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])

848	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.
849		
850		
851		
852	Cryptoki	The Cryptographic Token Interface defined in this standard.
853	Cryptoki library	A library that implements the functions specified in this standard.
854	CT-KIP	Cryptographic Token Key Initialization Protocol (as defined in [CT-KIP])
855		
856	DER	Distinguished Encoding Rules, as defined in X.690.
857	DES	Data Encryption Standard, as defined in [FIPS PUB 46-3].
858	DSA	Digital Signature Algorithm, as defined in [FIPS PUB 186-4].
859	EC	Elliptic Curve
860	ECB	Electronic Codebook mode, as defined in [FIPS PUB 81].
861	ECDH	Elliptic Curve Diffie-Hellman.
862	ECDSA	Elliptic Curve DSA, as in [ANSI X9.62].
863	ECMQV	Elliptic Curve Menezes-Qu-Vanstone
864	GOST 28147-89	The encryption algorithm, as defined in Part 2 [GOST 28147-89 and [RFC 4357] [RFC 4490], and RFC [4491].
865		
866	GOST R 34.11-94	Hash algorithm, as defined in [GOST R 34.11-94] and [RFC 4357], [RFC 4490], and [RFC 4491].
867		
868	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].
869		
870	IV	Initialization Vector.
871	KEM	Key Encapsulation Mechanism.
872	MAC	Message Authentication Code.
873	Mechanism	A process for implementing a cryptographic operation.
874	MQV	Menezes-Qu-Vanstone
875	OAEP	Optimal Asymmetric Encryption Padding for RSA.
876	Object	An item that is stored on a token. May be data, a certificate, or a key.
877		
878	PIN	Personal Identification Number.
879	PKCS	Public-Key Cryptography Standards.
880	PRF	Pseudo random function.
881	PTD	Personal Trusted Device, as defined in [MeT-PTD]
882	RSA	The RSA public-key cryptosystem.
883	Reader	The means by which information is exchanged with a device.
884	Session	A logical connection between an application and a token.
885	SHA-1	The (revised) Secure Hash Algorithm with a 160-bit message digest, as defined in [FIPS PUB 180-4].
886		
887	SHA-224	The Secure Hash Algorithm with a 224-bit message digest, as defined in [FIPS PUB 180-4].
888		

889	SHA-256	The Secure Hash Algorithm with a 256-bit message digest, as
890		defined in [FIPS PUB 180-4].
891	SHA-384	The Secure Hash Algorithm with a 384-bit message digest, as
892		defined in [FIPS PUB 180-4].
893	SHA-512	The Secure Hash Algorithm with a 512-bit message digest, as
894		defined in [FIPS PUB 180-4].
895	Slot	A logical reader that potentially contains a token.
896	SSL	The Secure Sockets Layer 3.0 protocol.
897	Subject Name	The X.500 distinguished name of the entity to which a key is
898		assigned.
899	SO	A Security Officer user.
900	TLS	Transport Layer Security.
901	Token	The logical view of a cryptographic device defined by Cryptoki.
902	User	The person using an application that interfaces to Cryptoki.
903	UTF-8	Universal Character Set (UCS) transformation format (UTF) that
904		represents ISO 10646 and UNICODE strings with a variable number
905		of octets.
906	WTLS	Wireless Transport Layer Security.

907 1.2 Symbols and abbreviations

908 The following symbols are used in this standard:

909 *Table 1, Symbols*

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

910 The following prefixes are used in this standard:

911 *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 [RFC 2279]. 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
```

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)(
```

```
1019     CK_VOID_PTR pReserved
1020 );
1021 myC_InitializeType myC_Initialize;
```

1022 ♦ CK_CALLBACK_FUNCTION

1023 CK_CALLBACK_FUNCTION(returnType, name), when followed by a parentheses-enclosed
1024 list of arguments and a semicolon, declares a variable or type which is a pointer to an application callback
1025 function that can be used by a Cryptoki API function in a Cryptoki library. returnType is the return type of
1026 the function, and name is its name. It SHALL be used in either of the following fashions to define a
1027 function pointer variable, myCallback, which can point to an application callback which takes arguments
1028 args and returns a CK_RV (note that neither of the following code snippets actually assigns a value to
1029 myCallback):

```
1030 CK_CALLBACK_FUNCTION(CK_RV, myCallback)(args);
1031
```

1032 or:

```
1033 typedef CK_CALLBACK_FUNCTION(CK_RV, myCallbackType)(args);
1034 myCallbackType myCallback;
```

1035 ♦ NULL_PTR

1036 NULL_PTR is the value of a NULL pointer. In any ANSI C environment—and in many others as well—
1037 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
3.1	0x03	0x01
3.2	0x03	0x02

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**.

1069 ♦ **CK_INFO; CK_INFO_PTR**

1070 **CK_INFO** provides general information about Cryptoki. It is defined as follows:

```
1071     typedef struct CK_INFO {  
1072         CK_VERSION cryptokiVersion;  
1073         CK_UTF8CHAR manufacturerID[32];  
1074         CK_FLAGS flags;  
1075         CK_UTF8CHAR libraryDescription[32];  
1076         CK_VERSION libraryVersion;  
1077     } CK_INFO;  
1078
```

1079 The fields of the structure have the following meanings:

1080	<i>cryptokiVersion</i>	Cryptoki interface version number, for compatibility with future
1081		revisions of this interface
1082	<i>manufacturerID</i>	ID of the Cryptoki library manufacturer. MUST be padded with the
1083		blank character (' '). Should <i>not</i> be null-terminated.
1084	<i>flags</i>	bit flags reserved for future versions. MUST be zero for this version
1085	<i>libraryDescription</i>	character-string description of the library. MUST be padded with the
1086		blank character (' '). Should <i>not</i> be null-terminated.
1087	<i>libraryVersion</i>	Cryptoki library version number

1088 For libraries written to this document, the value of *cryptokiVersion* should match the version of this
1089 specification; the value of *libraryVersion* is the version number of the library software itself.

1090 **CK_INFO_PTR** is a pointer to a **CK_INFO**.

1091 ♦ **CK_NOTIFICATION**

1092 **CK_NOTIFICATION** holds the types of notifications that Cryptoki provides to an application. It is defined
1093 as follows:

```
1094     typedef CK_ULONG CK_NOTIFICATION;  
1095
```

1096 For this version of Cryptoki, the following types of notifications are defined:

```
1097     CKN_SURRENDER  
1098
```

1099 The notifications have the following meanings:

1100	<i>CKN_SURRENDER</i>	Cryptoki is surrendering the execution of a function executing in a
1101		session so that the application may perform other operations. After
1102		performing any desired operations, the application should indicate
1103		to Cryptoki whether to continue or cancel the function (see Section
1104		5.22.1).

1105 **3.2 Slot and token types**

1106 Cryptoki represents slot and token information with the following types:

1107 ♦ **CK_SLOT_ID; CK_SLOT_ID_PTR**

1108 **CK_SLOT_ID** is a Cryptoki-assigned value that identifies a slot. It is defined as follows:

```
1109     typedef CK_ULONG CK_SLOT_ID;  
1110
```

1111 A list of **CK_SLOT_IDs** is returned by **C_GetSlotList**. A priori, *any* value of **CK_SLOT_ID** can be a valid
1112 slot identifier—in particular, a system may have a slot identified by the value 0. It need not have such a
1113 slot, however.
1114 **CK_SLOT_ID_PTR** is a pointer to a **CK_SLOT_ID**.

1115 ♦ **CK_SLOT_INFO; CK_SLOT_INFO_PTR**

1116 **CK_SLOT_INFO** provides information about a slot. It is defined as follows:

```
1117 typedef struct CK_SLOT_INFO {  
1118     CK_UTF8CHAR slotDescription[64];  
1119     CK_UTF8CHAR manufacturerID[32];  
1120     CK_FLAGS flags;  
1121     CK_VERSION hardwareVersion;  
1122     CK_VERSION firmwareVersion;  
1123 } CK_SLOT_INFO;  
1124
```

1125 The fields of the structure have the following meanings:

- 1126 *slotDescription* character-string description of the slot. MUST be padded with the
1127 blank character (‘ ’). MUST NOT be null-terminated.
- 1128 *manufacturerID* ID of the slot manufacturer. MUST be padded with the blank
1129 character (‘ ’). MUST NOT be null-terminated.
- 1130 *flags* bits flags that provide capabilities of the slot. The flags are defined
1131 below
- 1132 *hardwareVersion* version number of the slot’s hardware
- 1133 *firmwareVersion* version number of the slot’s firmware

1134 The following table defines the *flags* field:

1135 *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”

1136 Implementations should not imply an interpretation about the security properties based on the
1137 **CKF_HW_SLOT**. The flag is intended purely as an indicator that the slot is a physical realisation. How the
1138 implementation that offers the physical slot implements mechanisms (software/firmware/hardware) is
1139 unrelated to this flag.
1140 For a given slot, the value of the **CKF_REMOVABLE_DEVICE** flag *never changes*. In addition, if this flag
1141 is not set for a given slot, then the **CKF_TOKEN_PRESENT** flag for that slot is *always* set. That is, if a
1142 slot does not support a removable device, then that slot always has a token in it.
1143 **CK_SLOT_INFO_PTR** is a pointer to a **CK_SLOT_INFO**.

1144 ♦ **CK_TOKEN_INFO; CK_TOKEN_INFO_PTR**

1145 **CK_TOKEN_INFO** provides information about a token. It is defined as follows:

```

1146 typedef struct CK_TOKEN_INFO {
1147     CK_UTF8CHAR label[32];
1148     CK_UTF8CHAR manufacturerID[32];
1149     CK_UTF8CHAR model[16];
1150     CK_CHAR serialNumber[16];
1151     CK_FLAGS flags;
1152     CK_ULONG ulMaxSessionCount;
1153     CK_ULONG ulSessionCount;
1154     CK_ULONG ulMaxRwSessionCount;
1155     CK_ULONG ulRwSessionCount;
1156     CK_ULONG ulMaxPinLen;
1157     CK_ULONG ulMinPinLen;
1158     CK_ULONG ulTotalPublicMemory;
1159     CK_ULONG ulFreePublicMemory;
1160     CK_ULONG ulTotalPrivateMemory;
1161     CK_ULONG ulFreePrivateMemory;
1162     CK_VERSION hardwareVersion;
1163     CK_VERSION firmwareVersion;
1164     CK_CHAR utcTime[16];
1165 } CK_TOKEN_INFO;
1166

```

The fields of the structure have the following meanings:

1168	<i>label</i>	application-defined label, assigned during token initialization. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
1169		
1170		
1171	<i>manufacturerID</i>	ID of the device manufacturer. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
1172		
1173	<i>model</i>	model of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
1174		
1175	<i>serialNumber</i>	character-string serial number of the device. MUST be padded with the blank character (' '). MUST NOT be null-terminated.
1176		
1177	<i>flags</i>	bit flags indicating capabilities and status of the device as defined below
1178		
1179	<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)
1180		
1181		
1182	<i>ulSessionCount</i>	number of sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
1183		
1184	<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)
1185		
1186		
1187	<i>ulRwSessionCount</i>	number of read/write sessions that this application currently has open with the token (see CK_TOKEN_INFO Note below)
1188		
1189	<i>ulMaxPinLen</i>	maximum length in bytes of the PIN
1190	<i>ulMinPinLen</i>	minimum length in bytes of the PIN
1191	<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)
1192		
1193	<i>ulFreePublicMemory</i>	the amount of free (unused) memory on the token in bytes for public objects (see CK_TOKEN_INFO Note below)
1194		
1195	<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)
1196		

Bit Flag	Mask	Meaning
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.
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.
CKF_SEED_RANDOM_REQUIRED	0x02000000	True if the token's random number generator must be seeded or re-seeded using C_SeedRandom .
CKF_ASYNC_SESSION_SUPPORTED	0x04000000	True if the token supports asynchronous sessions (see Section 5.21).

- 1209 It is not specified in Cryptoki which type of random number generator, i.e., a true/physical or
1210 pseudo/deterministic or hybrid random number generator, the token has when **CKF_RNG** is true.
- 1211 Exactly what the **CKF_WRITE_PROTECTED** flag means is not specified in Cryptoki. An application may
1212 be unable to perform certain actions on a write-protected token; these actions can include any of the
1213 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_SO_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:

1266 ♦ **CK_SESSION_HANDLE; CK_SESSION_HANDLE_PTR**

1267 **CK_SESSION_HANDLE** is a Cryptoki-assigned value that identifies a session. It is defined as follows:

1268

```
typedef CK_ULONG CK_SESSION_HANDLE;
```

1270 *Valid session handles in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki
1271 defines the following symbolic value:

1272

```
CK_INVALID_HANDLE
```

1274 CK_SESSION_HANDLE_PTR is a pointer to a CK_SESSION_HANDLE.

1275 ♦ **CK_USER_TYPE**

1276 **CK_USER_TYPE** holds the types of Cryptoki users described in [\[PKCS11-UG\]](#) and, in addition, a
1277 context-specific type described in Section 4.10. It is defined as follows:

1278

```
typedef CK_ULONG CK_USER_TYPE;
```

1280 For this version of Cryptoki, the following types of users are defined:

1281

```
CKU_SO  
1282   CKU_USER  
1283   CKU_CONTEXT_SPECIFIC
```

1284 ♦ **CK_STATE**

1285 **CK_STATE** holds the session state, as described in [\[PKCS11-UG\]](#). It is defined as follows:

1286

```
typedef CK_ULONG CK_STATE;
```

1288 For this version of Cryptoki, the following session states are defined:

1289

```
CKS_RO_PUBLIC_SESSION  
1290   CKS_RO_USER_FUNCTIONS  
1291   CKS_RW_PUBLIC_SESSION  
1292   CKS_RW_USER_FUNCTIONS  
1293   CKS_RW_SO_FUNCTIONS
```

1294 ♦ **CK_SESSION_INFO; CK_SESSION_INFO_PTR**

1295 **CK_SESSION_INFO** provides information about a session. It is defined as follows:

1296

```
typedef struct CK_SESSION_INFO {  
1297     CK_SLOT_ID slotID;  
1298     CK_STATE state;  
1299     CK_FLAGS flags;  
1300     CK_ULONG ulDeviceError;  
1301   } CK_SESSION_INFO;  
1302
```

1303
1304 The fields of the structure have the following meanings:

1305 *slotID* ID of the slot that interfaces with the token
1306 *state* the state of the session

1307 *flags* bit flags that define the type of session; the flags are defined below
1308 *ulDeviceError* an error code defined by the cryptographic device. Used for errors
1309 not covered by Cryptoki.

1310 The following table defines the *flags* field:

1311 *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
CKF_ASYNC_SESSION	0x00000008	True if the session is asynchronous; false if the session is synchronous (see Section 5.21)

1312 CK_SESSION_INFO_PTR is a pointer to a CK_SESSION_INFO.

1313 **3.4 Object types**

1314 Cryptoki represents object information with the following types:

1315 **◆ CK_OBJECT_HANDLE; CK_OBJECT_HANDLE_PTR**

1316 **CK_OBJECT_HANDLE** is a token-specific identifier for an object. It is defined as follows:

```
1317 typedef CK_ULONG CK_OBJECT_HANDLE;  
1318
```

1319 When an object is created or found on a token by an application, Cryptoki assigns it an object handle for
1320 that application's sessions to use to access it. A particular object on a token does not necessarily have a
1321 handle which is fixed for the lifetime of the object; however, if a particular session can use a particular
1322 handle to access a particular object, then that session will continue to be able to use that handle to
1323 access that object as long as the session continues to exist, the object continues to exist, and the object
1324 continues to be accessible to the session.

1325 *Valid object handles in Cryptoki always have nonzero values.* For developers' convenience, Cryptoki
1326 defines the following symbolic value:

```
1327 CK_INVALID_HANDLE  
1328
```

1329 CK_OBJECT_HANDLE_PTR is a pointer to a CK_OBJECT_HANDLE.

1330 **◆ CK_OBJECT_CLASS; CK_OBJECT_CLASS_PTR**

1331 CK_OBJECT_CLASS is a value that identifies the classes (or types) of objects that Cryptoki recognizes.
1332 It is defined as follows:

```
1333 typedef CK_ULONG CK_OBJECT_CLASS;  
1334
```

1335 Object classes are defined with the objects that use them. The type is specified on an object through the
1336 **CKA_CLASS** attribute of the object.

1337 Vendor defined values for this type may also be specified.

```
1338 CKO_VENDOR_DEFINED  
1339
```

1340 Object classes **CKO_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
1341 interoperability, vendors should register their object classes through the PKCS process.

1342 **CK_OBJECT_CLASS_PTR** is a pointer to a **CK_OBJECT_CLASS**.

1343 ◆ **CK_HW_FEATURE_TYPE**

1344 **CK_HW_FEATURE_TYPE** is a value that identifies a hardware feature type of a device. It is defined as
1345 follows:

```
1346     typedef CK_ULONG CK_HW_FEATURE_TYPE;  
1347
```

1348 Hardware feature types are defined with the objects that use them. The type is specified on an object
1349 through the **CKA_HW_FEATURE_TYPE** attribute of the object.

1350 Vendor defined values for this type may also be specified.

```
1351     CKH_VENDOR_DEFINED  
1352
```

1353 Feature types **CKH_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
1354 interoperability, vendors should register their feature types through the PKCS process.

1355 ◆ **CK_KEY_TYPE**

1356 **CK_KEY_TYPE** is a value that identifies a key type. It is defined as follows:

```
1357     typedef CK_ULONG CK_KEY_TYPE;  
1358
```

1359 Key types are defined with the objects and mechanisms that use them. The key type is specified on an
1360 object through the **CKA_KEY_TYPE** attribute of the object.

1361 Vendor defined values for this type may also be specified.

```
1362     CKK_VENDOR_DEFINED  
1363
```

1364 Key types **CKK_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
1365 interoperability, vendors should register their key types through the PKCS process.

1366 ◆ **CK_CERTIFICATE_TYPE**

1367 **CK_CERTIFICATE_TYPE** is a value that identifies a certificate type. It is defined as follows:

```
1368     typedef CK_ULONG CK_CERTIFICATE_TYPE;  
1369
```

1370 Certificate types are defined with the objects and mechanisms that use them. The certificate type is
1371 specified on an object through the **CKA_CERTIFICATE_TYPE** attribute of the object.

1372 Vendor defined values for this type may also be specified.

```
1373     CKC_VENDOR_DEFINED  
1374
```

1375 Certificate types **CKC_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
1376 interoperability, vendors should register their certificate types through the PKCS process.

1377 ◆ **CK_CERTIFICATE_CATEGORY**

1378 **CK_CERTIFICATE_CATEGORY** is a value that identifies a certificate category. It is defined as follows:

1379

1380

```
typedef CK_ULONG CK_CERTIFICATE_CATEGORY;
```

1381

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

1382

Table 8, Certificate category values

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)

1383

◆ CK_ATTRIBUTE_TYPE

1384

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

1385

1386

```
typedef CK_ULONG CK_ATTRIBUTE_TYPE;
```

1387

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.

1389

Vendor defined values for this type may also be specified.

1390

1391

```
CKA_VENDOR_DEFINED
```

1392

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.

1394

◆ CK_ATTRIBUTE; CK_ATTRIBUTE_PTR

1395

1396

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

1397

1398

1399

1400

1401

1402

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

1403

The fields of the structure have the following meanings:

1404

1405

1406

<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

1407

1408

1409

1410

1411

1412

1413

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:

year the year ("1900" - "9999")

month the month ("01" - "12")

day 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. ID's 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**.

1453 ♦ **CK_JAVA_MIDP_SECURITY_DOMAIN**

1454 **CK_JAVA_MIDP_SECURITY_DOMAIN** is a value that identifies the Java MIDP security domain of a
1455 certificate. It is defined as follows:

1456

```
typedef CK_ULONG CK_JAVA_MIDP_SECURITY_DOMAIN;
```

1457 For this version of Cryptoki, the following security domains are defined. See the [Java MIDP] specification
1458 for further information:

1459 *Table 9, Security domain values*

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

1460

1461 **3.5 Data types for mechanisms**

1462 Cryptoki supports the following types for describing mechanisms and parameters to them:

1463 ♦ **CK_MECHANISM_TYPE; CK_MECHANISM_TYPE_PTR**

1464 **CK_MECHANISM_TYPE** is a value that identifies a mechanism type. It is defined as follows:

1465

```
typedef CK_ULONG CK_MECHANISM_TYPE;
```


1466

1467 Mechanism types are defined with the objects and mechanism descriptions that use them.
1468 Vendor defined values for this type may also be specified.

1469

```
CKM_VENDOR_DEFINED
```


1470

1471 Mechanism types **CKM_VENDOR_DEFINED** and above are permanently reserved for token vendors. For
1472 interoperability, vendors should register their mechanism types through the PKCS process.

1473 **CK_MECHANISM_TYPE_PTR** is a pointer to a **CK_MECHANISM_TYPE**.

1474 ♦ **CK_MECHANISM; CK_MECHANISM_PTR**

1475 **CK_MECHANISM** is a structure that specifies a particular mechanism and any parameters it requires. It
1476 is defined as follows:

1477

```
typedef struct CK_MECHANISM {  
1478     CK_MECHANISM_TYPE mechanism;  
1479     CK_VOID_PTR pParameter;  
1480     CK_ULONG ulParameterLen;  
1481 } CK_MECHANISM;
```


1482

1483 The fields of the structure have the following meanings:

1484 *mechanism*the type of mechanism
1485 *pParameter*pointer to the parameter if required by the mechanism
1486 *ulParameterLen*length in bytes of the parameter
1487 Note that *pParameter* is a “void” pointer, facilitating the passing of arbitrary values. Both the application
1488 and the Cryptoki library MUST ensure that the pointer can be safely cast to the expected type (*i.e.*,
1489 without word-alignment errors).
1490 **CK_MECHANISM_PTR** is a pointer to a **CK_MECHANISM**.

1491 ♦ **CK_MECHANISM_INFO; CK_MECHANISM_INFO_PTR**

1492 **CK_MECHANISM_INFO** is a structure that provides information about a particular mechanism. It is
1493 defined as follows:

```
1494 typedef struct CK_MECHANISM_INFO {  
1495     CK_ULONG ulMinKeySize;  
1496     CK_ULONG ulMaxKeySize;  
1497     CK_FLAGS flags;  
1498 } CK_MECHANISM_INFO;  
1499
```

1500 The fields of the structure have the following meanings:

- 1501 *ulMinKeySize* the minimum size of the key for the mechanism (whether this is
1502 measured in bits or in bytes is mechanism-dependent)
- 1503 *ulMaxKeySize* the maximum size of the key for the mechanism (whether this is
1504 measured in bits or in bytes is mechanism-dependent)
- 1505 *flags* bit flags specifying mechanism capabilities

1506 For some mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields have meaningless values.

1507 The following table defines the *flags* field:

1508 *Table 10, 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.

Bit Flag	Mask	Meaning
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 and C_VerifySignatureInit
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
CKF_ENCAPSULATE	0x10000000	True if the mechanism can be used with C_EncapsulateKey
CKF_DECAPSULATE	0x20000000	True if the mechanism can be used with C_DecapsulateKey
CKF_EXTENSION	0x80000000	True if there is an extension to the flags; false if no extensions. MUST be false for this version.

1509 CK_MECHANISM_INFO_PTR is a pointer to a CK_MECHANISM_INFO.

1510 **3.6 Function types**

1511 Cryptoki represents information about functions with the following data types:

1512 **◆ CK_RV**

1513 **CK_RV** is a value that identifies the return value of a Cryptoki function. It is defined as follows:

```
1514 typedef CK_ULONG CK_RV;  
1515
```

1516 Vendor defined values for this type may also be specified.

```
1517 CKR_VENDOR_DEFINED  
1518
```

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

◆ CK_NOTIFY

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

```
typedef CK_CALLBACK_FUNCTION(CK_RV, CK_NOTIFY) (  
    CK_SESSION_HANDLE hSession,  
    CK_NOTIFICATION event,  
    CK_VOID_PTR pApplication  
);
```

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

hSession The handle of the session performing the callback

event The type of notification callback

pApplication An application-defined value. This is the same value as was passed to **C_OpenSession** to open the session performing the callback

◆ CK_C_XXX

Cryptoki also defines an entire family of other function pointer types. For each function **C_XXX** in the Cryptoki API (see Section 5 for detailed information about each of them), Cryptoki defines a type **CK_C_XXX**, which is a pointer to a function with the same arguments and return value as **C_XXX** has. An appropriately-set variable of type **CK_C_XXX** may be used by an application to call the Cryptoki function **C_XXX**.

◆ CK_FUNCTION_LIST; CK_FUNCTION_LIST_PTR; CK_FUNCTION_LIST_PTR_PTR

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

```
typedef struct CK_FUNCTION_LIST {  
    CK_VERSION version;  
    CK_C_Initialize C_Initialize;  
    CK_C_Finalize C_Finalize;  
    CK_C_GetInfo C_GetInfo;  
    CK_C_GetFunctionList C_GetFunctionList;  
    CK_C_GetSlotList C_GetSlotList;  
    CK_C_GetSlotInfo C_GetSlotInfo;  
    CK_C_GetTokenInfo C_GetTokenInfo;  
    CK_C_GetMechanismList C_GetMechanismList;  
    CK_C_GetMechanismInfo C_GetMechanismInfo;  
    CK_C_InitToken C_InitToken;  
    CK_C_InitPIN C_InitPIN;  
    CK_C_SetPIN C_SetPIN;  
    CK_C_OpenSession C_OpenSession;  
    CK_C_CloseSession C_CloseSession;  
    CK_C_CloseAllSessions C_CloseAllSessions;  
    CK_C_GetSessionInfo C_GetSessionInfo;  
  
    CK_C_GetOperationState C_GetOperationState;  
    CK_C_SetOperationState C_SetOperationState;  
    CK_C_Login C_Login;  
    CK_C_Logout C_Logout;
```

```

1569 CK_C_CreateObject C_CreateObject;
1570 CK_C_CopyObject C_CopyObject;
1571 CK_C_DestroyObject C_DestroyObject;
1572 CK_C_GetObjectSize C_GetObjectSize;
1573 CK_C_GetAttributeValue C_GetAttributeValue;
1574 CK_C_SetAttributeValue C_SetAttributeValue;
1575 CK_C_FindObjectsInit C_FindObjectsInit;
1576 CK_C_FindObjects C_FindObjects;
1577 CK_C_FindObjectsFinal C_FindObjectsFinal;
1578 CK_C_EncryptInit C_EncryptInit;
1579 CK_C_Encrypt C_Encrypt;
1580 CK_C_EncryptUpdate C_EncryptUpdate;
1581 CK_C_EncryptFinal C_EncryptFinal;
1582 CK_C_DecryptInit C_DecryptInit;
1583 CK_C_Decrypt C_Decrypt;
1584 CK_C_DecryptUpdate C_DecryptUpdate;
1585 CK_C_DecryptFinal C_DecryptFinal;
1586 CK_C_DigestInit C_DigestInit;
1587 CK_C_Digest C_Digest;
1588 CK_C_DigestUpdate C_DigestUpdate;
1589 CK_C_DigestKey C_DigestKey;
1590 CK_C_DigestFinal C_DigestFinal;
1591 CK_C_SignInit C_SignInit;
1592 CK_C_Sign C_Sign;
1593 CK_C_SignUpdate C_SignUpdate;
1594 CK_C_SignFinal C_SignFinal;
1595 CK_C_SignRecoverInit C_SignRecoverInit;
1596 CK_C_SignRecover C_SignRecover;
1597 CK_C_VerifyInit C_VerifyInit;
1598 CK_C_Verify C_Verify;
1599 CK_C_VerifyUpdate C_VerifyUpdate;
1600 CK_C_VerifyFinal C_VerifyFinal;
1601 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1602 CK_C_VerifyRecover C_VerifyRecover;
1603 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1604 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1605 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1606 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1607 CK_C_GenerateKey C_GenerateKey;
1608 CK_C_GenerateKeyPair C_GenerateKeyPair;
1609 CK_C_WrapKey C_WrapKey;
1610 CK_C_UnwrapKey C_UnwrapKey;
1611 CK_C_DeriveKey C_DeriveKey;
1612 CK_C_SeedRandom C_SeedRandom;
1613 CK_C_GenerateRandom C_GenerateRandom;
1614 CK_C_GetFunctionStatus C_GetFunctionStatus;
1615 CK_C_CancelFunction C_CancelFunction;
1616 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1617 } CK_FUNCTION_LIST;
1618

```

1619 Each Cryptoki library has a static **CK_FUNCTION_LIST** structure, and a pointer to it (or to a copy of it
1620 which is also owned by the library) may be obtained by the **C_GetFunctionList** function (see Section
1621 5.2). The value that this pointer points to can be used by an application to quickly find out where the
1622 executable code for each function in the Cryptoki API is located. Every function in the Cryptoki API MUST
1623 have an entry point defined in the Cryptoki library's **CK_FUNCTION_LIST** structure. If a particular
1624 function in the Cryptoki API is not supported by a library, then the function pointer for that function in the
1625 library's **CK_FUNCTION_LIST** structure should point to a function stub which simply returns
1626 **CKR_FUNCTION_NOT_SUPPORTED**.

1627 In this structure 'version' is the cryptoki specification version number. The major and minor versions must
1628 be set to 0x02 and 0x28 indicating a version 2.40 compatible structure. The updated function list table for
1629 this version of the specification may be returned via **C_GetInterfaceList** or **C_GetInterface**.

1630

1631 An application may or may not be able to modify a Cryptoki library's static **CK_FUNCTION_LIST**

1632 structure. Whether or not it can, it should never attempt to do so.

1633 PKCS #11 modules must not add new functions at the end of the **CK_FUNCTION_LIST** that are not

1634 contained within the defined structure. If a PKCS #11 module needs to define additional functions, they

1635 should be placed within a vendor defined interface returned via **C_GetInterfaceList** or **C_GetInterface**.

1636 **CK_FUNCTION_LIST_PTR** is a pointer to a **CK_FUNCTION_LIST**.

1637 **CK_FUNCTION_LIST_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_PTR**.

1638

1639 ♦ **CK_FUNCTION_LIST_3_0; CK_FUNCTION_LIST_3_0_PTR;**

1640 **CK_FUNCTION_LIST_3_0_PTR_PTR**

1641 **CK_FUNCTION_LIST_3_0** is a structure which contains the same function pointers as in

1642 **CK_FUNCTION_LIST** and additional functions added to the end of the structure that were defined in

1643 Cryptoki version 3.0. It is defined as follows:

```
1644 typedef struct CK_FUNCTION_LIST_3_0 {
1645     CK_VERSION version;
1646     CK_C_Initialize C_Initialize;
1647     CK_C_Finalize C_Finalize;
1648     CK_C_GetInfo C_GetInfo;
1649     CK_C_GetFunctionList C_GetFunctionList;
1650     CK_C_GetSlotList C_GetSlotList;
1651     CK_C_GetSlotInfo C_GetSlotInfo;
1652     CK_C_GetTokenInfo C_GetTokenInfo;
1653     CK_C_GetMechanismList C_GetMechanismList;
1654     CK_C_GetMechanismInfo C_GetMechanismInfo;
1655     CK_C_InitToken C_InitToken;
1656     CK_C_InitPIN C_InitPIN;
1657     CK_C_SetPIN C_SetPIN;
1658     CK_C_OpenSession C_OpenSession;
1659     CK_C_CloseSession C_CloseSession;
1660     CK_C_CloseAllSessions C_CloseAllSessions;
1661     CK_C_GetSessionInfo C_GetSessionInfo;
1662     CK_C_GetOperationState C_GetOperationState;
1663     CK_C_SetOperationState C_SetOperationState;
1664     CK_C_Login C_Login;
1665     CK_C_Logout C_Logout;
1666     CK_C_CreateObject C_CreateObject;
1667     CK_C_CopyObject C_CopyObject;
1668     CK_C_DestroyObject C_DestroyObject;
1669     CK_C_GetObjectSize C_GetObjectSize;
1670     CK_C_GetAttributeValue C_GetAttributeValue;
1671     CK_C_SetAttributeValue C_SetAttributeValue;
1672     CK_C_FindObjectsInit C_FindObjectsInit;
1673     CK_C_FindObjects C_FindObjects;
1674     CK_C_FindObjectsFinal C_FindObjectsFinal;
1675     CK_C_EncryptInit C_EncryptInit;
1676     CK_C_Encrypt C_Encrypt;
1677     CK_C_EncryptUpdate C_EncryptUpdate;
1678     CK_C_EncryptFinal C_EncryptFinal;
1679     CK_C_DecryptInit C_DecryptInit;
1680     CK_C_Decrypt C_Decrypt;
1681     CK_C_DecryptUpdate C_DecryptUpdate;
1682     CK_C_DecryptFinal C_DecryptFinal;
1683     CK_C_DigestInit C_DigestInit;
1684     CK_C_Digest C_Digest;
1685     CK_C_DigestUpdate C_DigestUpdate;
1686     CK_C_DigestKey C_DigestKey;
```

```

1687 CK_C_DigestFinal C_DigestFinal;
1688 CK_C_SignInit C_SignInit;
1689 CK_C_Sign C_Sign;
1690 CK_C_SignUpdate C_SignUpdate;
1691 CK_C_SignFinal C_SignFinal;
1692 CK_C_SignRecoverInit C_SignRecoverInit;
1693 CK_C_SignRecover C_SignRecover;
1694 CK_C_VerifyInit C_VerifyInit;
1695 CK_C_Verify C_Verify;
1696 CK_C_VerifyUpdate C_VerifyUpdate;
1697 CK_C_VerifyFinal C_VerifyFinal;
1698 CK_C_VerifyRecoverInit C_VerifyRecoverInit;
1699 CK_C_VerifyRecover C_VerifyRecover;
1700 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1701 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1702 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1703 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1704 CK_C_GenerateKey C_GenerateKey;
1705 CK_C_GenerateKeyPair C_GenerateKeyPair;
1706 CK_C_WrapKey C_WrapKey;
1707 CK_C_UnwrapKey C_UnwrapKey;
1708 CK_C_DeriveKey C_DeriveKey;
1709 CK_C_SeedRandom C_SeedRandom;
1710 CK_C_GenerateRandom C_GenerateRandom;
1711 CK_C_GetFunctionStatus C_GetFunctionStatus;
1712 CK_C_CancelFunction C_CancelFunction;
1713 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1714 CK_C_GetInterfaceList C_GetInterfaceList;
1715 CK_C_GetInterface C_GetInterface;
1716 CK_C_LoginUser C_LoginUser;
1717 CK_C_SessionCancel C_SessionCancel;
1718 CK_C_MessageEncryptInit C_MessageEncryptInit;
1719 CK_C_EncryptMessage C_EncryptMessage;
1720 CK_C_EncryptMessageBegin C_EncryptMessageBegin;
1721 CK_C_EncryptMessageNext C_EncryptMessageNext;
1722 CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1723 CK_C_MessageDecryptInit C_MessageDecryptInit;
1724 CK_C_DecryptMessage C_DecryptMessage;
1725 CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1726 CK_C_DecryptMessageNext C_DecryptMessageNext;
1727 CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1728 CK_C_MessageSignInit C_MessageSignInit;
1729 CK_C_SignMessage C_SignMessage;
1730 CK_C_SignMessageBegin C_SignMessageBegin;
1731 CK_C_SignMessageNext C_SignMessageNext;
1732 CK_C_MessageSignFinal C_MessageSignFinal;
1733 CK_C_MessageVerifyInit C_MessageVerifyInit;
1734 CK_C_VerifyMessage C_VerifyMessage;
1735 CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1736 CK_C_VerifyMessageNext C_VerifyMessageNext;
1737 CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1738 } CK_FUNCTION_LIST_3_0;
1739

```

1740 For a general description of **CK_FUNCTION_LIST_3_0** see **CK_FUNCTION_LIST**.

1741 In this structure, *version* is the cryptoki specification version number. It should match the value of
1742 *cryptokiVersion* returned in the **CK_INFO** structure, but must be 3.0 at minimum.

1743 This function list may be returned via **C_GetInterfaceList** or **C_GetInterface**

1744 **CK_FUNCTION_LIST_3_0_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0**.

1745 **CK_FUNCTION_LIST_3_0_PTR_PTR** is a pointer to a **CK_FUNCTION_LIST_3_0_PTR**.

1746 ♦ **CK_FUNCTION_LIST_3_2; CK_FUNCTION_LIST_3_2_PTR;**
1747 **CK_FUNCTION_LIST_3_2_PTR_PTR**

1748 **CK_FUNCTION_LIST_3_2** is a structure which contains the same function pointers as in
1749 **CK_FUNCTION_LIST_3_0** and additional functions added to the end of the structure that were defined in
1750 Cryptoki version 3.2. It is defined as follows:

```
1751 typedef struct CK_FUNCTION_LIST_3_2 {  
1752     CK_VERSION version;  
1753     CK_C_Initialize C_Initialize;  
1754     CK_C_Finalize C_Finalize;  
1755     CK_C_GetInfo C_GetInfo;  
1756     CK_C_GetFunctionList C_GetFunctionList;  
1757     CK_C_GetSlotList C_GetSlotList;  
1758     CK_C_GetSlotInfo C_GetSlotInfo;  
1759     CK_C_GetTokenInfo C_GetTokenInfo;  
1760     CK_C_GetMechanismList C_GetMechanismList;  
1761     CK_C_GetMechanismInfo C_GetMechanismInfo;  
1762     CK_C_InitToken C_InitToken;  
1763     CK_C_InitPIN C_InitPIN;  
1764     CK_C_SetPIN C_SetPIN;  
1765     CK_C_OpenSession C_OpenSession;  
1766     CK_C_CloseSession C_CloseSession;  
1767     CK_C_CloseAllSessions C_CloseAllSessions;  
1768     CK_C_GetSessionInfo C_GetSessionInfo;  
1769     CK_C_GetOperationState C_GetOperationState;  
1770     CK_C_SetOperationState C_SetOperationState;  
1771     CK_C_Login C_Login;  
1772     CK_C_Logout C_Logout;  
1773     CK_C_CreateObject C_CreateObject;  
1774     CK_C_CopyObject C_CopyObject;  
1775     CK_C_DestroyObject C_DestroyObject;  
1776     CK_C_GetObjectSize C_GetObjectSize;  
1777     CK_C_GetAttributeValue C_GetAttributeValue;  
1778     CK_C_SetAttributeValue C_SetAttributeValue;  
1779     CK_C_FindObjectsInit C_FindObjectsInit;  
1780     CK_C_FindObjects C_FindObjects;  
1781     CK_C_FindObjectsFinal C_FindObjectsFinal;  
1782     CK_C_EncryptInit C_EncryptInit;  
1783     CK_C_Encrypt C_Encrypt;  
1784     CK_C_EncryptUpdate C_EncryptUpdate;  
1785     CK_C_EncryptFinal C_EncryptFinal;  
1786     CK_C_DecryptInit C_DecryptInit;  
1787     CK_C_Decrypt C_Decrypt;  
1788     CK_C_DecryptUpdate C_DecryptUpdate;  
1789     CK_C_DecryptFinal C_DecryptFinal;  
1790     CK_C_DigestInit C_DigestInit;  
1791     CK_C_Digest C_Digest;  
1792     CK_C_DigestUpdate C_DigestUpdate;  
1793     CK_C_DigestKey C_DigestKey;  
1794     CK_C_DigestFinal C_DigestFinal;  
1795     CK_C_SignInit C_SignInit;  
1796     CK_C_Sign C_Sign;  
1797     CK_C_SignUpdate C_SignUpdate;  
1798     CK_C_SignFinal C_SignFinal;  
1799     CK_C_SignRecoverInit C_SignRecoverInit;  
1800     CK_C_SignRecover C_SignRecover;  
1801     CK_C_VerifyInit C_VerifyInit;  
1802     CK_C_Verify C_Verify;  
1803     CK_C_VerifyUpdate C_VerifyUpdate;  
1804     CK_C_VerifyFinal C_VerifyFinal;  
1805     CK_C_VerifyRecoverInit C_VerifyRecoverInit;  
1806     CK_C_VerifyRecover C_VerifyRecover;
```

```

1807 CK_C_DigestEncryptUpdate C_DigestEncryptUpdate;
1808 CK_C_DecryptDigestUpdate C_DecryptDigestUpdate;
1809 CK_C_SignEncryptUpdate C_SignEncryptUpdate;
1810 CK_C_DecryptVerifyUpdate C_DecryptVerifyUpdate;
1811 CK_C_GenerateKey C_GenerateKey;
1812 CK_C_GenerateKeyPair C_GenerateKeyPair;
1813 CK_C_WrapKey C_WrapKey;
1814 CK_C_UnwrapKey C_UnwrapKey;
1815 CK_C_DeriveKey C_DeriveKey;
1816 CK_C_SeedRandom C_SeedRandom;
1817 CK_C_GenerateRandom C_GenerateRandom;
1818 CK_C_GetFunctionStatus C_GetFunctionStatus;
1819 CK_C_CancelFunction C_CancelFunction;
1820 CK_C_WaitForSlotEvent C_WaitForSlotEvent;
1821 CK_C_GetInterfaceList C_GetInterfaceList;
1822 CK_C_GetInterface C_GetInterface;
1823 CK_C_LoginUser C_LoginUser;
1824 CK_C_SessionCancel C_SessionCancel;
1825 CK_C_MessageEncryptInit C_MessageEncryptInit;
1826 CK_C_EncryptMessage C_EncryptMessage;
1827 CK_C_EncryptMessageBegin C_EncryptMessageBegin;
1828 CK_C_EncryptMessageNext C_EncryptMessageNext;
1829 CK_C_MessageEncryptFinal C_MessageEncryptFinal;
1830 CK_C_MessageDecryptInit C_MessageDecryptInit;
1831 CK_C_DecryptMessage C_DecryptMessage;
1832 CK_C_DecryptMessageBegin C_DecryptMessageBegin;
1833 CK_C_DecryptMessageNext C_DecryptMessageNext;
1834 CK_C_MessageDecryptFinal C_MessageDecryptFinal;
1835 CK_C_MessageSignInit C_MessageSignInit;
1836 CK_C_SignMessage C_SignMessage;
1837 CK_C_SignMessageBegin C_SignMessageBegin;
1838 CK_C_SignMessageNext C_SignMessageNext;
1839 CK_C_MessageSignFinal C_MessageSignFinal;
1840 CK_C_MessageVerifyInit C_MessageVerifyInit;
1841 CK_C_VerifyMessage C_VerifyMessage;
1842 CK_C_VerifyMessageBegin C_VerifyMessageBegin;
1843 CK_C_VerifyMessageNext C_VerifyMessageNext;
1844 CK_C_MessageVerifyFinal C_MessageVerifyFinal;
1845 CK_C_EncapsulateKey C_EncapsulateKey;
1846 CK_C_DecapsulateKey C_DecapsulateKey;
1847 CK_C_VerifySignatureInit C_VerifySignatureInit;
1848 CK_C_VerifySignature C_VerifySignature;
1849 CK_C_VerifySignatureUpdate C_VerifySignatureUpdate;
1850 CK_C_VerifySignatureFinal C_VerifySignatureFinal;
1851 CK_C_GetSessionValidationFlags C_GetSessionValidationFlags;
1852 CK_C_AsyncComplete C_AsyncComplete;
1853 CK_C_AsyncGetID C_AsyncGetID;
1854 CK_C_AsyncJoin C_AsyncJoin;
1855 CK_C_WrapKeyAuthenticated C_WrapKeyAuthenticated;
1856 CK_C_UnwrapKeyAuthenticated C_UnwrapKeyAuthenticated;
1857 } CK_FUNCTION_LIST_3_2;
1858

```

For a general description of **CK_FUNCTION_LIST_3_2** see **CK_FUNCTION_LIST**.

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 3.2 at minimum.

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

CK_FUNCTION_LIST_3_2_PTR is a pointer to a **CK_FUNCTION_LIST_3_2**.

CK_FUNCTION_LIST_3_2_PTR_PTR is a pointer to a **CK_FUNCTION_LIST_3_2_PTR**.

1865 ♦ **CK_INTERFACE; CK_INTERFACE_PTR;**
1866 **CK_INTERFACE_PTR_PTR**

1867 **CK_INTERFACE** is a structure which contains an interface name with a function list and flag.
1868 It is defined as follows:

```
1869 typedef struct CK_INTERFACE {  
1870     CK_UTF8CHAR_PTR pInterfaceName;  
1871     CK_VOID_PTR      pFunctionList;  
1872     CK_FLAGS         flags;  
1873 } CK_INTERFACE;
```

1874
1875 The fields of the structure have the following meanings:

- 1876 *pInterfaceName* the name of the interface
- 1877 *pFunctionList* the interface function list which must always begin with a
1878 CK_VERSION structure as the first field
- 1879 *flags* bit flags specifying interface capabilities

1880 The interface name “PKCS 11” is reserved for use by interfaces defined within the cryptoki specification.
1881 Interfaces starting with the string: “Vendor ” are reserved for vendor use and will not oetherwise be
1882 defined as interfaces in the PKCS #11 specification. Vendors should supply new functions with interface
1883 names of “Vendor {vendor name}”. For example “Vendor ACME Inc”.

1884
1885 The following table defines the flags field:
1886 *Table 11, 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.

1887
1888 **CK_INTERFACE_PTR** is a pointer to a **CK_INTERFACE**.
1889 **CK_INTERFACE_PTR_PTR** is a pointer to a **CK_INTERFACE_PTR**.

1890 ♦ **CK_ASYNC_DATA, CK_ASYNC_DATA_PTR**

1891 **CK_ASYNC_DATA** is a structure used by asynchronous function management functions. It is defined as
1892 follows:

```
1893 typedef struct CK_ASYNC_DATA {  
1894     CK_ULONG ulVersion;  
1895     CK_BYTE_PTR pValue;  
1896     CK_ULONG ulValue;  
1897     CK_OBJECT_HANDLE hObject;  
1898     CK_OBJECT_HANDLE hAdditionalObject;
```

1899

1900

```
} CK_ASYNC_DATA;
```

1901

The fields of the structure have the following meanings:

1902

1903

1904

1905

1906

1907

1908

<i>ulVersion</i>	version of this structure; always 0 for this version of Cryptoki
<i>pValue</i>	on completion contains a pointer to the original input buffer, caller is responsible for this memory
<i>ulValue</i>	size of the result
<i>hObject</i>	receives the handle for an object resulting from the operation
<i>hAdditionalObject</i>	receives the handle for an additional object resulting from the operation

1909

CK_ASYNC_DATA_PTR is a pointer to a **CK_ASYNC_DATA**.

1910

1911

3.7 Locking-related types

1912

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.*

1913

1914

◆ **CK_CREATEMUTEX**

1915

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:

1916

1917

1918

1919

1920

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

1921

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:

1922

1923

1924

```
CKR_OK, CKR_GENERAL_ERROR  
CKR_HOST_MEMORY
```

1925

◆ **CK_DESTROYMUTEX**

1926

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

1927

1928

1929

1930

1931

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

1932

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:

1933

1934

1935

1936

```
CKR_OK, CKR_GENERAL_ERROR  
CKR_HOST_MEMORY  
CKR_MUTEX_BAD
```

1937

◆ CK_LOCKMUTEX and CK_UNLOCKMUTEX

1938

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

1939

1941

- If a **CK_LOCKMUTEX** function is called on a mutex which is not locked, the calling thread obtains a lock on that mutex and returns.

1942

1943

- If a **CK_LOCKMUTEX** function is called on a mutex which is locked by some thread other than the calling thread, the calling thread blocks and waits for that mutex to be unlocked.

1944

1945

- If a **CK_LOCKMUTEX** function is called on a mutex which is locked by the calling thread, the behavior of the function call is undefined.

1946

1947

- If a **CK_UNLOCKMUTEX** function is called on a mutex which is locked by the calling thread, that mutex is unlocked and the function call returns. Furthermore:

1948

1949

- If exactly one thread was blocking on that particular mutex, then that thread stops blocking, obtains a lock on that mutex, and its **CK_LOCKMUTEX** call returns.

1950

1951

- If more than one thread was blocking on that particular mutex, then exactly one of the blocking threads is selected somehow. That lucky thread stops blocking, obtains a lock on the mutex, and its **CK_LOCKMUTEX** call returns. All other threads blocking on that particular mutex continue to block.

1952

1953

1954

1955

- If a **CK_UNLOCKMUTEX** function is called on a mutex which is not locked, then the function call returns the error code **CKR_MUTEX_NOT_LOCKED**.

1956

1957

- If a **CK_UNLOCKMUTEX** function is called on a mutex which is locked by some thread other than the calling thread, the behavior of the function call is undefined.

1958

1959

CK_LOCKMUTEX is defined as follows:

1960

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

1961

1962

1963

1964

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

1965

1966

```
CKR_OK, CKR_GENERAL_ERROR
```

1967

```
CKR_HOST_MEMORY,
```

1968

```
CKR_MUTEX_BAD
```

1969

1970

CK_UNLOCKMUTEX is defined as follows:

1971

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

1972

1973

1974

1975

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

1976

1977

```
CKR_OK, CKR_GENERAL_ERROR
```

1978

```
CKR_HOST_MEMORY
```

1979

```
CKR_MUTEX_BAD
```

1980

```
CKR_MUTEX_NOT_LOCKED
```

1981 ♦ **CK_C_INITIALIZE_ARGS; CK_C_INITIALIZE_ARGS_PTR**

1982 **CK_C_INITIALIZE_ARGS** is a structure containing the optional arguments for the **C_Initialize** function.

1983 For this version of Cryptoki, these optional arguments are all concerned with the way the library deals

1984 with threads. **CK_C_INITIALIZE_ARGS** is defined as follows:

```
1985 typedef struct CK_C_INITIALIZE_ARGS {
1986     CK_CREATEMUTEX CreateMutex;
1987     CK_DESTROYMUTEX DestroyMutex;
1988     CK_LOCKMUTEX LockMutex;
1989     CK_UNLOCKMUTEX UnlockMutex;
1990     CK_FLAGS flags;
1991     CK_VOID_PTR pReserved;
1992 } CK_C_INITIALIZE_ARGS;
1993
```

1994 The fields of the structure have the following meanings:

1995 *CreateMutex* pointer to a function to use for creating mutex objects

1996 *DestroyMutex* pointer to a function to use for destroying mutex objects

1997 *LockMutex* pointer to a function to use for locking mutex objects

1998 *UnlockMutex* pointer to a function to use for unlocking mutex objects

1999 *flags* bit flags specifying options for **C_Initialize**; the flags are defined

2000 below

2001 *pReserved* reserved for future use. Should be **NULL_PTR** for this version of

2002 Cryptoki

2003 The following table defines the flags field:

2004 *Table 12, 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

2005 **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:

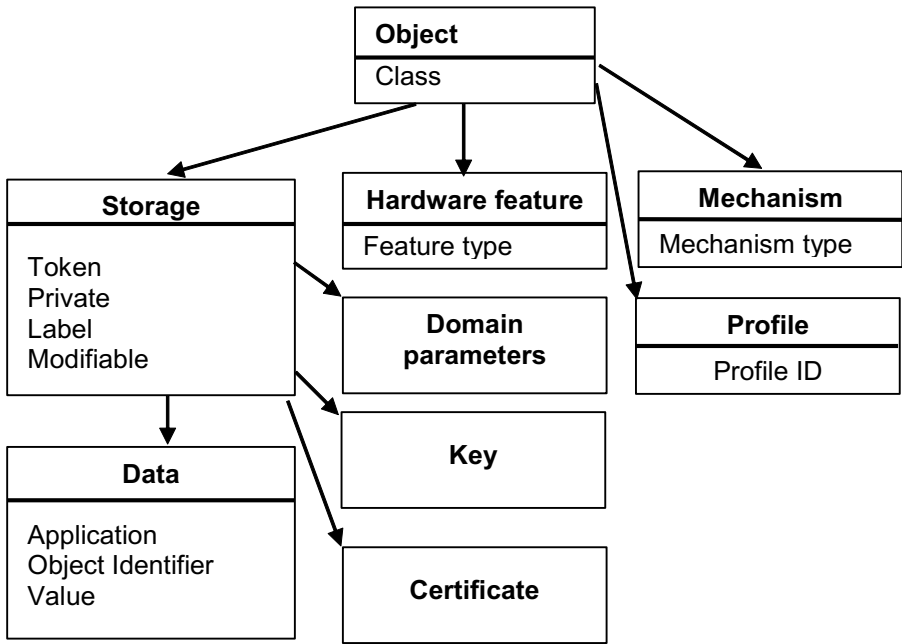


Figure 1, Object Attribute Hierarchy

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 section 6 Mechanisms 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 5.6.11), **C_GenerateKey**, **C_GenerateKeyPair**, **C_UnwrapKey**, **C_DeriveKey**, **C_EncapsulateKey**, and **C_DecapsulateKey** (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 5.6.11). 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 19) attribute is set to CK_FALSE, it may be copied with the Cryptoki function **C_CopyObject** (see Section 5.6.11). 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 13, 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.
¹³ Attribute is generated by the module with **C_GenerateKeyPair**.

Table 14, Common Object Attributes

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

¹ Refer to Table 13 for footnotes

The above table defines the attributes common to all objects.

4.3 Hardware Feature Objects

4.3.1 Definitions

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

4.3.2 Overview

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

Table 15, Hardware Feature Common Attributes

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

¹ Refer to Table 13 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 16, 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 17, 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 18, 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.

2181 Table 19, Common Storage Object Attributes

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.

2182 Only the **CKA_LABEL** attribute can be modified after the object is created. (The **CKA_TOKEN**,
2183 **CKA_PRIVATE**, and **CKA_MODIFIABLE** attributes can be changed in the process of copying an object,
2184 however.)

2185 The **CKA_TOKEN** attribute identifies whether the object is a token object or a session object.

2186 When the **CKA_PRIVATE** attribute is CK_TRUE, a user may not access the object until the user has
2187 been authenticated to the token.

2188 The value of the **CKA_MODIFIABLE** attribute determines whether or not an object is read-only.

2189 The **CKA_LABEL** attribute is intended to assist users in browsing.

2190 The value of the **CKA_COPYABLE** attribute determines whether or not an object can be copied. This
2191 attribute can be used in conjunction with **CKA_MODIFIABLE** to prevent changes to the permitted usages
2192 of keys and other objects.

2193 The value of the **CKA_DESTROYABLE** attribute determines whether the object can be destroyed using
2194 **C_DestroyObject**.

2195 **4.4.1 The CKA_UNIQUE_ID attribute**

2196 Any time a new object is created, a value for **CKA_UNIQUE_ID** MUST be generated by the token and
2197 stored with the object. The specific algorithm used to generate unique ID values for objects is token-
2198 specific, but values generated MUST be unique across all objects visible to any particular session, and
2199 SHOULD be unique across all objects created by the token. Reinitializing the token, such as by calling
2200 **C_InitToken**, MAY cause reuse of **CKA_UNIQUE_ID** values.

2201 Any attempt to modify the **CKA_UNIQUE_ID** attribute of an existing object or to specify the value of the
2202 **CKA_UNIQUE_ID** attribute in the template for an operation that creates one or more objects MUST fail.
2203 Operations failing for this reason return the error code **CKR_ATTRIBUTE_READ_ONLY**.

2204

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 20, 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:

2243 Table 21, 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)

2244 Refer to Table 13 for footnotes

2245 Cryptoki does not enforce the relationship of the **CKA_PUBLIC_KEY_INFO** to the public key in the
2246 certificate, but does recommend that the key be extracted from the certificate to create this value.

2247 The **CKA_CERTIFICATE_TYPE** attribute may not be modified after an object is created. This version of
2248 Cryptoki supports the following certificate types:

- 2249
 - X.509 public key certificate
 - 2250 • WTLS public key certificate
 - 2251 • X.509 attribute certificate

2252 The **CKA_TRUSTED** attribute cannot be set to CK_TRUE by an application. It MUST be set by a token
2253 initialization application or by the token's SO. Trusted certificates cannot be modified.

2254 The **CKA_CERTIFICATE_CATEGORY** attribute is used to indicate if a stored certificate is a user
2255 certificate for which the corresponding private key is available on the token ("token user"), a CA certificate
2256 ("authority"), or another end-entity certificate ("other entity"). This attribute may not be modified after an
2257 object is created.

2258 The **CKA_CERTIFICATE_CATEGORY** and **CKA_TRUSTED** attributes will together be used to map to
2259 the categorization of the certificates.

2260 **CKA_CHECK_VALUE**: The value of this attribute is derived from the certificate by taking the first three
2261 bytes of the SHA-1 hash of the certificate object's **CKA_VALUE** attribute.

2262 The **CKA_START_DATE** and **CKA_END_DATE** attributes are for reference only; Cryptoki does not
2263 attach any special meaning to them. When present, the application is responsible to set them to values
2264 that match the certificate's encoded "not before" and "not after" fields (if any).

2265 **4.6.3 X.509 public key certificate objects**

2266 X.509 certificate objects (certificate type **CKC_X_509**) hold X.509 public key certificates. The following
2267 table defines the X.509 certificate object attributes, in addition to the common attributes defined for this
2268 object class:

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.

2270 ¹MUST be specified when the object is created.

2271 ²MUST be specified when the object is created. MUST be non-empty if **CKA_URL** is empty.

2272 ³MUST be non-empty if **CKA_VALUE** is empty.

2273 ⁴Can only be empty if **CKA_URL** is empty.

2274 Only the **CKA_ID**, **CKA_ISSUER**, and **CKA_SERIAL_NUMBER** attributes may be modified after the

2275 object is created.

2276 The **CKA_ID** attribute is intended as a means of distinguishing multiple public-key/private-key pairs held

2277 by the same subject (whether stored in the same token or not). (Since the keys are distinguished by

2278 subject name as well as identifier, it is possible that keys for different subjects may have the same

2279 **CKA_ID** value without introducing any ambiguity.)

2280 It is intended in the interests of interoperability that the subject name and key identifier for a certificate will

2281 be the same as those for the corresponding public and private keys (though it is not required that all be

2282 stored in the same token). However, Cryptoki does not enforce this association, or even the uniqueness

2283 of the key identifier for a given subject; in particular, an application may leave the key identifier empty.

2284 The **CKA_ISSUER** and **CKA_SERIAL_NUMBER** attributes are for compatibility with [PKCS #7] and

2285 Privacy Enhanced Mail [RFC 1421]. Note that with the version 3 extensions to X.509 certificates, the key

2286 identifier may be carried in the certificate. It is intended that the **CKA_ID** value be identical to the key

2287 identifier in such a certificate extension, although this will not be enforced by Cryptoki.

2288 The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found
2289 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile
2290 environments.

2291 The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY**
2292 attributes are used to store the hashes of the public keys of the subject and the issuer. They are
2293 particularly important when only the URL is available to be able to correlate a certificate with a private key
2294 and when searching for the certificate of the issuer. The hash algorithm is defined by
2295 **CKA_NAME_HASH_ALGORITHM**.

2296 The **CKA_JAVA_MIDP_SECURITY_DOMAIN** attribute associates a certificate with a Java MIDP security
2297 domain.

2298 The following is a sample template for creating an X.509 certificate object:

```
2299 CK_OBJECT_CLASS class = CKO_CERTIFICATE;  
2300 CK_CERTIFICATE_TYPE certType = CKC_X_509;  
2301 CK_UTF8CHAR label[] = "A certificate object";  
2302 CK_BYTE subject[] = {...};  
2303 CK_BYTE id[] = {123};  
2304 CK_BYTE certificate[] = {...};  
2305 CK_BBOOL true = CK_TRUE;  
2306 CK_ATTRIBUTE template[] = {  
2307     {CKA_CLASS, &class, sizeof(class)},  
2308     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};  
2309     {CKA_TOKEN, &true, sizeof(true)},  
2310     {CKA_LABEL, label, sizeof(label)-1},  
2311     {CKA_SUBJECT, subject, sizeof(subject)},  
2312     {CKA_ID, id, sizeof(id)},  
2313     {CKA_VALUE, certificate, sizeof(certificate)}  
2314 };
```

2315 **4.6.4 WTLS public key certificate objects**

2316 WTLS certificate objects (certificate type **CKC_WTLS**) hold WTLS public key certificates. The following
2317 table defines the WTLS certificate object attributes, in addition to the common attributes defined for this
2318 object class.

2319 *Table 23, 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_KEY and CKA_HASH_OF_ISSUER_PUBLIC_KEY. If the attribute is not present then the type defaults to SHA-1.

2320 ¹MUST be specified when the object is created. Can only be empty if **CKA_VALUE** is empty.

2321 ²MUST be specified when the object is created. MUST be non-empty if **CKA_URL** is empty.

2322 ³MUST be non-empty if **CKA_VALUE** is empty.

2323 ⁴Can only be empty if **CKA_URL** is empty.

2324

2325 Only the **CKA_ISSUER** attribute may be modified after the object has been created.

2326 The encoding for the **CKA_SUBJECT**, **CKA_ISSUER**, and **CKA_VALUE** attributes can be found in

2327 [WTLS].

2328 The **CKA_URL** attribute enables the support for storage of the URL where the certificate can be found

2329 instead of the certificate itself. Storage of a URL instead of the complete certificate is often used in mobile

2330 environments.

2331 The **CKA_HASH_OF_SUBJECT_PUBLIC_KEY** and **CKA_HASH_OF_ISSUER_PUBLIC_KEY**

2332 attributes are used to store the hashes of the public keys of the subject and the issuer. They are

2333 particularly important when only the URL is available to be able to correlate a certificate with a private key

2334 and when searching for the certificate of the issuer. The hash algorithm is defined by

2335 **CKA_NAME_HASH_ALGORITHM**.

2336 The following is a sample template for creating a WTLS certificate object:

```

2337 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
2338 CK_CERTIFICATE_TYPE certType = CKC_WTLS;
2339 CK_UTF8CHAR label[] = "A certificate object";
2340 CK_BYTE subject[] = {...};
2341 CK_BYTE certificate[] = {...};
2342 CK_BBOOL true = CK_TRUE;
2343 CK_ATTRIBUTE template[] =
2344 {
2345     {CKA_CLASS, &class, sizeof(class)},
2346     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)},
2347     {CKA_TOKEN, &true, sizeof(true)},
2348     {CKA_LABEL, label, sizeof(label)-1},
2349     {CKA_SUBJECT, subject, sizeof(subject)},
2350     {CKA_VALUE, certificate, sizeof(certificate)}
2351 };

```

2352 4.6.5 X.509 attribute certificate objects

2353 X.509 attribute certificate objects (certificate type **CKC_X_509_ATTR_CERT**) hold X.509 attribute

2354 certificates. The following table defines the X.509 attribute certificate object attributes, in addition to the

2355 common attributes defined for this object class:

2356 Table 24, 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.

2357 ¹MUST be specified when the object is created

2358 Only the **CKA_AC_ISSUER**, **CKA_SERIAL_NUMBER** and **CKA_ATTR_TYPES** attributes may be

2359 modified after the object is created.

2360 The following is a sample template for creating an X.509 attribute certificate object:

```
2361 CK_OBJECT_CLASS class = CKO_CERTIFICATE;
2362 CK_CERTIFICATE_TYPE certType = CKC_X_509_ATTR_CERT;
2363 CK_UTF8CHAR label[] = "An attribute certificate object";
2364 CK_BYTE owner[] = {...};
2365 CK_BYTE certificate[] = {...};
2366 CK_BBOOL true = CK_TRUE;
2367 CK_ATTRIBUTE template[] = {
2368     {CKA_CLASS, &class, sizeof(class)},
2369     {CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)};
2370     {CKA_TOKEN, &true, sizeof(true)},
2371     {CKA_LABEL, label, sizeof(label)-1},
2372     {CKA_OWNER, owner, sizeof(owner)},
2373     {CKA_VALUE, certificate, sizeof(certificate)}
2374 };
```

2375 **4.7 Trust objects**

2376 **4.7.1 Definitions**

2377 This section defines the object class **CKO_TRUST** for type CK_OBJECT_CLASS as used in the

2378 **CKA_CLASS** attribute of objects.

2379

2380 CK_TRUST is defined as:

```
2381     typedef CK_ULONG CK_TRUST;
```

2382

2383 and can have the following values: CKT_TRUSTED, CKT_TRUST_ANCHOR, CKT_NOT_TRUSTED,

2384 CKT_TRUST_MUST_VERIFY_TRUST, or CKT_TRUST_UNKNOWN.

4.7.2 Overview

Trust objects (object class **CKO_TRUST**) bind trusted usages to individual certificates. Trust objects for a given certificate are accessed using the **CKA_ISSUER** and **CKA_SERIAL_NUMBER** attributes, and may be confirmed by comparing **CKA_HASH_OF_CERTIFICATE** with a recomputed hash value. The corresponding certificate does not necessarily have to exist in the same token as its trust object. Multiple trust objects for the same certificate can exist in different tokens, but each token should have no more than one trust object for a given certificate.

Table 25, Trust Object Attributes

Attribute	Data Type	Meaning
CKA_ISSUER ¹	Byte array	DER-encoding of the certificate issuer name (default empty)
CKA_SERIAL_NUMBER ¹	Byte array	DER-encoding of the certificate serial number (default empty)
CKA_HASH_OF_CERTIFICATE ²	Byte array	cryptographic hash of the certificate computed by CKA_NAME_HASH_ALGORITHM (default empty)
CKA_NAME_HASH_ALGORITHM ²	CK_MECHANISM_TYPE	mechanism used to calculate CKA_HASH_OF_CERTIFICATE (defaults to SHA-1 if not present)
CKA_TRUST_SERVER_AUTH ³	CK_TRUST	trust for authenticating the server in a client/server interaction (as in TLS/SSL/SSH)
CKA_TRUST_CLIENT_AUTH ³	CK_TRUST	trust for authenticating the client in a client/server interaction (as in TLS/SSL/SSH)
CKA_TRUST_CODE_SIGNING ³	CK_TRUST	trust for authenticating a code fragment
CKA_TRUST_EMAIL_PROTECTION ³	CK_TRUST	trust for authenticating an email user
CKA_TRUST_IPSEC_IKE ³	CK_TRUST	trust for IPSEC
CKA_TRUST_TIME_STAMPING ³	CK_TRUST	trust for Timestamping
CKA_TRUST_OCSP_SIGNING ³	CK_TRUST	trust for OCSP Signing

¹MUST be specified when the object is created.

²MUST be specified when the object is created unless all trust attributes are **CKT_TRUST_UNKNOWN**, or **CKT_NOT_TRUSTED**.

³Missing **CKA_TRUST_XXX** attributes are treated as **CKT_TRUST_UNKNOWN**.

CKA_TRUST_XXX attributes map roughly to Certificate EKU values, and carry the same semantics. If **CKA_MODIFIABLE** is not set in the template, it defaults to **CK_TRUE**; if **CKA_PRIVATE** is not set in the template, it defaults to **CK_FALSE**.

To obtain the effective trust attributes for a given certificate, the typical application will first:

1. identify the tokens containing a Trust object with matching **CKA_ISSUER** and **CKA_SERIAL_NUMBER** (and optionally check that **CKA_HASH_OF_CERTIFICATE** agrees with the hash of the certificate computed using **CKA_NAME_HASH_ALGORITHM**),
2. determine which of those Trust objects should be processed (presumably according to an established security policy), and
3. arrange the selected Trust objects in a list sorted in order of increasing priority.

Now, taking the first Trust object in the list as the initial working Trust object (WTO) with all omitted attributes assumed to have the value **CKT_TRUST_UNKNOWN**, the remaining Trust objects in the list are iteratively merged into it as follows:

- if the value of a trust attribute in the current object is **CKT_TRUST_UNKNOWN**, that attribute is left unchanged in the WTO,
- otherwise, the current attribute value replaces the attribute value in the WTO .

2414 Note that at any step of this process, an attribute value of **CKT_TRUST_MUST_VERIFY_TRUST** in the
2415 current Trust object resets any trust or distrust assigned to that attribute in the WTO by a lower priority
2416 token.
2417 When the process is complete, the final (“effective”) trust attribute values are to be interpreted as follows:

CKT_TRUSTED	the certificate is trusted for the associated operation
CKT_TRUST_ANCHOR	the certificate is trusted as a root signing certificate for chain validation of a cert that is trusted for the associate operation; this applies even when the certificate is not self-signed and when the certificate does not have the proper attributes to be CA certificate
CKT_NOT_TRUSTED	the certificate is explicitly not trusted for the associated operation, nor can trust chain through the certificate to an otherwise trusted root; this attribute can be used to ‘revoke’ intermediate CA certificates that have been compromised without removing trust from the parent certificate
CKT_TRUST_MUST_VERIFY_TRUST CKT_TRUST_UNKNOWN	the certificate is neither trusted nor untrusted for the associated operation

2418 Note that when processing a certificate chain, applications may use the various Trust objects to override
2419 trust attributes that would otherwise be associated with each certificate solely based on EKUs and other
2420 extensions encountered along the chain.
2421 The following is a sample template for creating an X.509 certificate object:

```
2422 CK_OBJECT_CLASS class = CKO_CERTIFICATE;  
2423 CK_UTF8CHAR label[] = "A certificate object";  
2424 CK_BYTE issuer[] = {...}; // matches certificate's issuer  
2425 CK_BYTE serialNumber[] = {...}; // matches certificate's serialNumber  
2426 CK_BYTE certificate[] = {...};  
2427 CK_BBOOL true = CK_TRUE;  
2428 CK_TRUST trustAnchor = CKT_TRUST_ANCHOR;  
2429 CK_TRUST notTrusted = CKT_NOT_TRUSTED;  
2430 CK_MECHANISM_TYPE hashMec = CKM_SHA265  
2431 CK_ATTRIBUTE template[] = {  
2432     {CKA_CLASS, &class, sizeof(class)},  
2433     {CKA_TOKEN, &>true, sizeof(true)},  
2434     {CKA_LABEL, label, sizeof(label)-1},  
2435     {CKA_ISSUER, issuer, sizeof(issuer)},  
2436     {CKA_SERIAL_NUBMER, serialNumber, sizeof(serialNumber)},  
2437     {CKA_HASH_OF_CERTIIFICATE, hash(hashMec,certificate,  
2438     sizeof(certificate),hashLen(hashMec)),  
2439     {CKA_NAME_HASH_ALGORITHM, &hashMech, sizeof(hashMech)},  
2440     {CKA_TRUST_SERVER_AUTH, &trustAnchor, sizeof(trustAnchor) },  
2441     {CKA_TRUST_CODE_SIGNING, &notTrusted, sizeof(notTrusted) }  
2442     // other attributes are CKT_TRUST_UNKNOWN if not included here.  
2443 };
```

2444 **4.8 Key objects**

2445 **4.8.1 Definitions**

2446 There is no **CKO_** definition for the base key object class, only for the key types derived from it.
2447 This section defines the object class **CKO_PUBLIC_KEY**, **CKO_PRIVATE_KEY** and
2448 **CKO_SECRET_KEY** for type **CK_OBJECT_CLASS** as used in the **CKA_CLASS** attribute of objects.

4.8.2 Overview

Key objects hold encryption or authentication keys, which can be public keys, private keys, or secret keys. The following common footnotes apply to all the tables describing attributes of keys:

The following table defines the attributes common to public key, private key and secret key classes, in addition to the common attributes defined for this object class:

Table 26, Common Key Attributes

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 callcreated 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.
CKA_OBJECT_VALIDATION_FLAGS ^{4,6,9,12}	CK_FLAGS	Object was created consistent with the validations appearing in flags.

Refer to Table 13 for footnotes

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

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

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

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

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

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

4.9 Public key objects

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

Table 27, Common Public Key Attributes

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.e., can be used to wrap other keys) ⁹
CKA_ENCAPSULATE ⁸	CK_BBOOL	CK_TRUE if key supports encapsulation (i.e., can be used in a KEM to create an encapsulated key and ciphertext for C_DecapsulateKey) 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)
CKA_PUBLIC_CRC64_VALUE ^{1,4,13}	Byte array	The CRC-64-ECMA calculated over the public key object's CKA_VALUE attribute unless otherwise specified in the mechanism description

Refer to Table 13 for footnotes

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

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

Table 28, 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

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

```
SubjectPublicKeyInfo ::= SEQUENCE {
    algorithm           AlgorithmIdentifier,
    subjectPublicKey     BIT_STRING }
```

The encodings for the subjectPublicKey field are specified in the description of the public key types in the appropriate sections for the key types defined within this specification.

4.10 Private key objects

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

Table 29, 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.e., can be used to unwrap other keys) ⁹
CKA_DECAPSULATE ⁸	CK_BBOOL	CK_TRUE if key supports decapsulation ⁹
CKA_EXTRACTABLE ^{8,12}	CK_BBOOL	CK_TRUE if key is extractable and can be wrapped ⁹

Attribute	Data type	Meaning
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. 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.
CKA_PUBLIC_CRC64_VALUE ^{1,4,13}	Byte array	The CRC-64-ECMA calculated over the public key object's CKA_VALUE attribute unless otherwise specified in the mechanism description

2494 Refer to Table 13 for footnotes

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

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

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

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

4.11 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 30, 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

Attribute	Data type	Meaning
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
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

Attribute	Data type	Meaning
		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 ulValueLen component of the attribute divided by the size of CK_ATTRIBUTE.

2539 Refer to Table 13 for footnotes

2540 If the **CKA_SENSITIVE** attribute is CK_TRUE, or if the **CKA_EXTRACTABLE** attribute is CK_FALSE,
2541 then certain attributes of the secret key cannot be revealed in plaintext outside the token. Which attributes
2542 these are is specified for each type of secret key in the attribute table in the section describing that type of
2543 key.

2544 The key check value (KCV) attribute for symmetric key objects to be called **CKA_CHECK_VALUE**, of
2545 type byte array, length 3 bytes, operates like a fingerprint, or checksum of the key. They are intended to
2546 be used to cross-check symmetric keys against other systems where the same key is shared, and as a
2547 validity check after manual key entry or restore from backup. Refer to object definitions of specific key
2548 types for KCV algorithms.

2549 Properties:

2550 1. For two keys that are cryptographically identical the value of this attribute should be identical.

2551 2. **CKA_CHECK_VALUE** should not be usable to obtain any part of the key value.

2552 3. Non-uniqueness. Two different keys can have the same **CKA_CHECK_VALUE**. This is unlikely
2553 (the probability can easily be calculated) but possible.

2554 The attribute is optional, but if supported, regardless of how the key object is created or derived, the value
2555 of the attribute is always supplied. It SHALL be supplied even if the encryption operation for the key is
2556 forbidden (i.e. when **CKA_ENCRYPT** is set to CK_FALSE).

2557 If a value is supplied in the application template (allowed but never necessary) then, if supported, it MUST
2558 match what the library calculates it to be or the library returns a **CKR_ATTRIBUTE_VALUE_INVALID**. If
2559 the library does not support the attribute then it should ignore it. Allowing the attribute in the template this
2560 way does no harm and allows the attribute to be treated like any other attribute for the purposes of key
2561 wrap and unwrap where the attributes are preserved also.

2562 The generation of the KCV may be prevented by the application supplying the attribute in the template as
2563 a no-value (0 length) entry. The application can query the value at any time like any other attribute using
2564 **C_GetAttributeValue**. **C_SetAttributeValue** may be used to destroy the attribute, by supplying no-value.

2565 Unless otherwise specified for the object definition, the value of this attribute is derived from the key
2566 object by taking the first three bytes of an encryption of a single block of null (0x00) bytes, using the
2567 default cipher and mode (e.g. ECB) associated with the key type of the secret key object.

2568 4.12 Domain parameter objects

2569 4.12.1 Definitions

2570 This section defines the object class **CKO_DOMAIN_PARAMETERS** for type CK_OBJECT_CLASS as
2571 used in the **CKA_CLASS** attribute of objects.

4.12.2 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 31, 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_GenerateKeycreated 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 13 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.

4.13 Mechanism objects

4.13.1 Definitions

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

4.13.2 Overview

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

Table 32, 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.

4.14 Profile objects

4.14.1 Definitions

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

4.14.2 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 33, Profile Object Attributes

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

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

4.15 Validation objects

4.15.1 Definitions

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

4.15.2 Overview

Validation objects (object class **CKO_VALIDATION**) describe which third party validations the module conforms to. Validation objects are read only, token objects.

Table 34, Validation Object Attributes

Attribute	Data Type	Meaning
CKA_VALIDATION_TYPE	CK_VALIDATION_TYPE	Identifier indicating the validation type
CKA_VALIDATION_VERSION	CK_VERSION	Version of the validation standard or specification
CKA_VALIDATION_LEVEL	CK_ULONG	Validation level, Meaning is Validation type specific
CKA_VALIDATION_MODULE_ID	CK_UTF8CHAR	How the module is identified in the validation documentation
CKA_VALIDATION_FLAG	CK_FLAGS	Flags identifying this validation in sessions and objects
CKA_VALIDATION_AUTHORITY_TYPE	CK_VALIDATION_AUTHORITY_TYPE	Identifies the authority that issues the validation
CKA_VALIDATION_COUNTRY ¹	CK_UTF8CHAR	2 letter ISO country code
CKA_VALIDATION_CERTIFICATE_IDENTIFIER ¹	CK_UTF8CHAR	Identifier of the validation certificate
CKA_VALIDATION_CERTIFICATE_URI ¹	CK_UTF8CHAR	Validation authority URI from which information related to the validation is available. If the Validation Certificate URI is not provided, the validation object SHOULD include a Validation Vendor URI.
CKA_VALIDATION_VENDOR_URI ¹	CK_UTF8CHAR	Validation Vendor URI from which information related to the validation is available.
CKA_VALIDATION_PROFILE ¹	CK_UTF8CHAR	Profile used for validation

¹ Optional value; may be empty.

2614

2615 ♦ **CK_VALIDATION_TYPE, CK_VALIDATION_TYPE_PTR**

2616 **CK_VALIDATION_TYPE** identifies the type of validation. It is defined as follows:

```
2617     typedef CK_ULONG CK_VALIDATION_TYPE;  
2618     typedef CK_VALIDATION_TYPE CK_PTR CK_VALIDATION_TYPE_PTR;
```

2619 Valid values are:

```
2620     CKV_TYPE_UNSPECIFIED  
2621     CKV_TYPE_SOFTWARE  
2622     CKV_TYPE_HARDWARE  
2623     CKV_TYPE_FIRMWARE  
2624     CKV_TYPE_HYBRID
```

2625 ♦ **CK_VALIDATION_AUTHORITY_TYPE,
2626 CK_VALIDATION_AUTHORITY_TYPE_PTR**

2627 **CK_VALIDATION_AUTHORITY_TYPE** identifies the authority that issues the validation. It is defined as
2628 follows:

```
2629     typedef CK_ULONG CK_VALIDATION_AUTHORITY_TYPE;  
2630     typedef CK_VALIDATION_AUTHORITY_TYPE CK_PTR  
2631     CK_VALIDATION_AUTHORITY_TYPE_PTR;
```

2632 Valid values are:

```
2633     CKV_AUTHORITY_TYPE_UNSPECIFIED  
2634     CKV_AUTHORITY_TYPE_NIST_CMVP  
2635     CKV_AUTHORITY_TYPE_COMMON_CRITERIA
```

2636 **4.15.3 Validation Indicators**

2637 Validation indicators are runtime indicators if a particular operation meets the appropriate criteria for this
2638 module running under the given validation rules. These rules will vary by validation type and even by
2639 different modules using various validation types.

2640 **4.15.3.1 Session validation flags**

2641 Sessions carry validation flags. These can be queried with **C_GetSessionValidationFlags**. Session
2642 validation flags are defined as follows:

```
2643     typedef CK_ULONG CK_SESSION_VALIDATION_FLAGS_TYPE;
```

2644 Valid values are:

```
2645     CKS_LAST_VALIDATION_OK
```

2646 **Last operation flags**

2647 The last operation flag is set if the last operation that completed (the last **C_XXXFinal**, or the last single
2648 short operation **C_WrapKey**, **C_DeriveKey**, etc.) met all the requirements of a validated mechanism.
2649 This allows access to the state of operations that don't return a key object.

2650 **4.15.3.2 Key object state**

2651 **CKA_OBJECT_VALIDATION_FLAGS** can only be set in ways conforming to the module's validation.
2652 Key objects typically take on the flags of the operation that created them, but are subject to the modules
2653 requirements under the module's validation.

2654

2655 Application notes:

2656 Applications should be prepared for changes in semantics as various validations change their guidance.

2657 Many of these operation chains, so in SSL, if the final key objects have the appropriate flags set in
2658 **CKA_OBJECT_VALIDATION_FLAGS**, then it generally means that all the operations (unwrap,
2659 key_derive, etc.) occurred in a manner that matches the module's validation, so the application would
2660 generally only need to query the validation flags of the final keys.

5 Functions

Cryptoki's functions are organized into the following categories:

- general-purpose functions (6 functions)
- slot and token management functions (9 functions)
- session management functions (11 functions)
- object management functions (9 functions)
- encryption functions (4 functions)
- message-based encryption functions (5 functions)
- decryption functions (4 functions)
- message-based decryption functions (5 functions)
- message digesting functions (5 functions)
- signing and MACing functions (6 functions)
- message-based signing functions (5 functions)
- functions for verifying signatures and MACs (10 functions)
- message-based functions for verifying signatures and MACs (5 functions)
- dual-purpose cryptographic functions (4 functions)
- key management functions (9 functions)
- random number generation functions (2 functions)
- parallel function management functions (2 functions)
- asynchronous function management functions (3 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 35, 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

Category	Function	Description
	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
	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
	C_GetSessionValidationFlags	fetches validation flags from the session
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

Category	Function	Description
Message-based Decryption Functions	C_MessageDecryptInit	initializes a message decryption operation
	C_DecryptMessage	decrypts single-part encrypted data
	C_DecryptMessageBegin	starts a multiple-part message decryption operation
	C_DecryptMessageNext	Continues and finishes a multiple-part message decryption operation
	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
	C_VerifySignatureInit	initializes a verification operation, passing the signature at initialization time.
	C_VerifySignature	verifies a signature on single-part data
	C_VerifySignatureUpdate	continues a multiple-part verification operation
	C_VerifySignatureFinal	finishes a multiple-part verification operation
Message-based Functions for verifying signatures and MACs	C_MessageVerifyInit	initializes a message verification operation
	C_VerifyMessage	verifies a signature on single-part data
	C_VerifyMessageBegin	starts a multiple-part message verification operation
	C_VerifyMessageNext	continues and finishes a multiple-part message verification operation

Category	Function	Description
	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 functions	C_GenerateKey	generates a secret key
	C_GenerateKeyPair	generates a public-key/private-key pair
	C_WrapKey	wraps (encrypts) a key
	C_UnwrapKey	unwraps (decrypts) a key
	C_DeriveKey	derives a key from a base key
	C_WrapKeyAuthenticated	authenticated key wrapping (encryption) of a key
	C_UnwrapKeyAuthenticated	authenticated key unwrapping (decryption) of a key
	C_EncapsulateKey	generates a secret key from a public key and returns the encapsulated ciphertext (KEM)
	C_DecapsulateKey	generates a secret key from a private key and the previously encapsulated ciphertext(KEM)
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
Asynchronous function management functions	C_AsyncComplete	checks if an asynchronous function has completed
	C_AsyncGetID	persist an asynchronous operation past a C_Finalize call
	C_AsyncJoin	reconnects the client application to an asynchronous operation
Callback function		application-supplied function to process notifications from Cryptoki

2687

2688 Execution of a Cryptoki function call is in general an all-or-nothing affair, *i.e.*, a function call accomplishes
2689 either its entire goal, or nothing at all.

2690 • If a Cryptoki function executes successfully, it returns the value **CKR_OK**.

2691 • If a Cryptoki function does not execute successfully, it returns some value other than **CKR_OK**, and
2692 the token is in the same state as it was in prior to the function call. If the function call was supposed to
2693 modify the contents of certain memory addresses on the host computer, these memory addresses
2694 may have been modified, despite the failure of the function.

2695 • Some Cryptoki function may return **CKR_PENDING** to indicate that function execution has not
2696 finished yet. For details see Section 5.21.

2697 • In unusual (and extremely unpleasant!) circumstances, a function can fail with the return value
 2698 **CKR_GENERAL_ERROR**. When this happens, the token and/or host computer may be in an
 2699 inconsistent state, and the goals of the function may have been partially achieved.

2700 There are a small number of Cryptoki functions whose return values do not behave precisely as
 2701 described above; these exceptions are documented individually with the description of the functions
 2702 themselves.

2703 A Cryptoki library need not support every function in the Cryptoki API. However, even an unsupported
 2704 function **MUST** have a “stub” in the library which simply returns the value
 2705 **CKR_FUNCTION_NOT_SUPPORTED**. The function’s entry in the library’s **CK_FUNCTION_LIST**
 2706 structure (as obtained by **C_GetFunctionList**) should point to this stub function (see Section 3.6).

2707 5.1 Function return values

2708 The Cryptoki interface possesses a large number of functions and return values. In Section 5.1, we
 2709 enumerate the various possible return values for Cryptoki functions; most of the remainder of Section 5.1
 2710 details the behavior of Cryptoki functions, including what values each of them may return.

2711 Because of the complexity of the Cryptoki specification, it is recommended that Cryptoki applications
 2712 attempt to give some leeway when interpreting Cryptoki functions’ return values. We have attempted to
 2713 specify the behavior of Cryptoki functions as completely as was feasible; nevertheless, there are
 2714 presumably some gaps. For example, it is possible that a particular error code which might apply to a
 2715 particular Cryptoki function is unfortunately not actually listed in the description of that function as a
 2716 possible error code. It is conceivable that the developer of a Cryptoki library might nevertheless permit
 2717 his/her implementation of that function to return that error code. It would clearly be somewhat ungraceful
 2718 if a Cryptoki application using that library were to terminate by abruptly dumping core upon receiving that
 2719 error code for that function. It would be far preferable for the application to examine the function’s return
 2720 value, see that it indicates some sort of error (even if the application doesn’t know precisely *what* kind of
 2721 error), and behave accordingly.

2722 See Section 5.1.8 for some specific details on how a developer might attempt to make an application that
 2723 accommodates a range of behaviors from Cryptoki libraries.

2724 5.1.1 Universal Cryptoki function return values

2725 Any Cryptoki function can return any of the following values:

- 2726 • **CKR_GENERAL_ERROR**: Some horrible, unrecoverable error has occurred. In the worst case, it is
 2727 possible that the function only partially succeeded, and that the computer and/or token is in an
 2728 inconsistent state.
- 2729 • **CKR_HOST_MEMORY**: The computer that the Cryptoki library is running on has insufficient memory
 2730 to perform the requested function.
- 2731 • **CKR_FUNCTION_FAILED**: The requested function could not be performed, but detailed information
 2732 about why not is not available in this error return. If the failed function uses a session, it is possible
 2733 that the **CK_SESSION_INFO** structure that can be obtained by calling **C_GetSessionInfo** will hold
 2734 useful information about what happened in its *ulDeviceError* field. In any event, although the function
 2735 call failed, the situation is not necessarily totally hopeless, as it is likely to be when
 2736 **CKR_GENERAL_ERROR** is returned. Depending on what the root cause of the error actually was, it
 2737 is possible that an attempt to make the exact same function call again would succeed.
- 2738 • **CKR_OPERATION_NOT_VALIDATED**: The requested operation violates one or more of the token’s
 2739 validation policies. Tokens may choose to return a more specific error (like
 2740 **CKR_ATTRIBUTE_VALUE_INVALID** or **CKR_DATA_LEN_RANGE**).
- 2741 • **CKR_OK**: The function executed successfully. Technically, **CKR_OK** is not *quite* a “universal” return
 2742 value; in particular, the legacy functions **C_GetFunctionStatus** and **C_CancelFunction** (see Section
 2743 5.20) cannot return **CKR_OK**.

2744 The relative priorities of these errors are in the order listed above, *e.g.*, if either of
2745 CKR_GENERAL_ERROR or CKR_HOST_MEMORY would be an appropriate error return, then
2746 CKR_GENERAL_ERROR should be returned.

2747 5.1.2 Cryptoki function return values for functions that use a session 2748 handle

2749 Any Cryptoki function that takes a session handle as one of its arguments (*i.e.*, any Cryptoki function
2750 except for **C_Initialize**, **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**,
2751 **C_GetTokenInfo**, **C_WaitForSlotEvent**, **C_GetMechanismList**, **C_GetMechanismInfo**, **C_InitToken**,
2752 **C_OpenSession**, and **C_CloseAllSessions**) can return the following values:

- 2753 • CKR_SESSION_HANDLE_INVALID: The specified session handle was invalid *at the time that the*
2754 *function was invoked*. Note that this can happen if the session's token is removed before the function
2755 invocation, since removing a token closes all sessions with it.
- 2756 • CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.
- 2757 • CKR_SESSION_CLOSED: The session was closed *during the execution of the function*. Note that,
2758 as stated in **[PKCS11-UG]**, the behavior of Cryptoki is *undefined* if multiple threads of an application
2759 attempt to access a common Cryptoki session simultaneously. Therefore, there is actually no
2760 guarantee that a function invocation could ever return the value CKR_SESSION_CLOSED. An
2761 example of multiple threads accessing a common session simultaneously is where one thread is
2762 using a session when another thread closes that same session.

2763 The relative priorities of these errors are in the order listed above, *e.g.*, if either of
2764 CKR_SESSION_HANDLE_INVALID or CKR_DEVICE_REMOVED would be an appropriate error return,
2765 then CKR_SESSION_HANDLE_INVALID should be returned.

2766 In practice, it is often not crucial (or possible) for a Cryptoki library to be able to make a distinction
2767 between a token being removed *before* a function invocation and a token being removed *during* a
2768 function execution.

2769 5.1.3 Cryptoki function return values for functions that use a token

2770 Any Cryptoki function that uses a particular token (*i.e.*, any Cryptoki function except for **C_Initialize**,
2771 **C_Finalize**, **C_GetInfo**, **C_GetFunctionList**, **C_GetSlotList**, **C_GetSlotInfo**, or **C_WaitForSlotEvent**)
2772 can return any of the following values:

- 2773 • CKR_DEVICE_MEMORY: The token does not have sufficient memory to perform the requested
2774 function.
- 2775 • CKR_DEVICE_ERROR: Some problem has occurred with the token and/or slot. This error code can
2776 be returned by more than just the functions mentioned above; in particular, it is possible for
2777 **C_GetSlotInfo** to return CKR_DEVICE_ERROR.
- 2778 • CKR_TOKEN_NOT_PRESENT: The token was not present in its slot *at the time that the function was*
2779 *invoked*.
- 2780 • CKR_DEVICE_REMOVED: The token was removed from its slot *during the execution of the function*.
- 2781 • CKR_TOKEN_NOT_INITIALIZED: The token is in factory state and **C_InitToken** (or an out of band
2782 initialization method) needs to be called before any other token related operations can be completed.

2783 The relative priorities of these errors are in the order listed above, *e.g.*, if either of
2784 CKR_DEVICE_MEMORY or CKR_DEVICE_ERROR would be an appropriate error return, then
2785 CKR_DEVICE_MEMORY should be returned.

2786 In practice, it is often not critical (or possible) for a Cryptoki library to be able to make a distinction
2787 between a token being removed *before* a function invocation and a token being removed *during* a
2788 function execution.

5.1.4 Special return value for application-supplied callbacks

There is a special-purpose return value which is not returned by any function in the actual Cryptoki API, but which may be returned by an application-supplied callback function. It is:

- **CKR_CANCEL**: When a function executing in serial with an application decides to give the application a chance to do some work, it calls an application-supplied function with a **CKN_SURRENDER** callback (see Section 5.21). If the callback returns the value **CKR_CANCEL**, then the function aborts and returns **CKR_FUNCTION_CANCELED**.

5.1.5 Special return values for mutex-handling functions

There are two other special-purpose return values which are not returned by any actual Cryptoki functions. These values may be returned by application-supplied mutex-handling functions, and they may safely be ignored by application developers who are not using their own threading model. They are:

- **CKR_MUTEX_BAD**: This error code can be returned by mutex-handling functions that are passed a bad mutex object as an argument. Unfortunately, it is possible for such a function not to recognize a bad mutex object. There is therefore no guarantee that such a function will successfully detect bad mutex objects and return this value.
- **CKR_MUTEX_NOT_LOCKED**: This error code can be returned by mutex-unlocking functions. It indicates that the mutex supplied to the mutex-unlocking function was not locked.

5.1.6 All other Cryptoki function return values

Descriptions of the other Cryptoki function return values follow. Except as mentioned in the descriptions of particular error codes, there are in general no particular priorities among the errors listed below, *i.e.*, if more than one error code might apply to an execution of a function, then the function may return any applicable error code.

- **CKR_ACTION_PROHIBITED**: This value can only be returned by **C_CopyObject**, **C_SetAttributeValue** and **C_DestroyObject**. It denotes that the action may not be taken, either because of underlying policy restrictions on the token, or because the object has the relevant **CKA_COPYABLE**, **CKA_MODIFIABLE** or **CKA_DESTROYABLE** policy attribute set to **CK_FALSE**.
- **CKR_ARGUMENTS_BAD**: This is a rather generic error code which indicates that the arguments supplied to the Cryptoki function were in some way not appropriate.
- **CKR_ATTRIBUTE_READ_ONLY**: An attempt was made to set a value for an attribute which may not be set by the application, or which may not be modified by the application. See Section 4.1 for more information.
- **CKR_ATTRIBUTE_SENSITIVE**: An attempt was made to obtain the value of an attribute of an object which cannot be satisfied because the object is either sensitive or un-extractable.
- **CKR_ATTRIBUTE_TYPE_INVALID**: An invalid attribute type was specified in a template. See Section 4.1 for more information.
- **CKR_ATTRIBUTE_VALUE_INVALID**: An invalid value was specified for a particular attribute in a template. See Section 4.1 for more information.
- **CKR_BUFFER_TOO_SMALL**: The output of the function is too large to fit in the supplied buffer.
- **CKR_CANT_LOCK**: This value can only be returned by **C_Initialize**. It means that the type of locking requested by the application for thread-safety is not available in this library, and so the application cannot make use of this library in the specified fashion.
- **CKR_CRYPTOKI_ALREADY_INITIALIZED**: This value can only be returned by **C_Initialize**. It means that the Cryptoki library has already been initialized (by a previous call to **C_Initialize** which did not have a matching **C_Finalize** call).
- **CKR_CRYPTOKI_NOT_INITIALIZED**: This value can be returned by any function other than **C_Initialize**, **C_GetFunctionList**, **C_GetInterfaceList** and **C_GetInterface**. It indicates that the

2835 function cannot be executed because the Cryptoki library has not yet been initialized by a call to
 2836 **C_Initialize**.

- 2837 • **CKR_CURVE_NOT_SUPPORTED**: This curve is not supported by this token. Used with Elliptic
 2838 Curve mechanisms.
- 2839 • **CKR_DATA_INVALID**: The plaintext input data to a cryptographic operation is invalid. This return
 2840 value has lower priority than **CKR_DATA_LEN_RANGE**.
- 2841 • **CKR_DATA_LEN_RANGE**: The plaintext input data to a cryptographic operation has a bad length.
 2842 Depending on the operation's mechanism, this could mean that the plaintext data is too short, too
 2843 long, or is not a multiple of some particular block size. This return value has higher priority than
 2844 **CKR_DATA_INVALID**.
- 2845 • **CKR_DOMAIN_PARAMS_INVALID**: Invalid or unsupported domain parameters were supplied to the
 2846 function. Which representation methods of domain parameters are supported by a given mechanism
 2847 can vary from token to token.
- 2848 • **CKR_ENCRYPTED_DATA_INVALID**: The encrypted input to a decryption operation has been
 2849 determined to be invalid ciphertext. This return value has lower priority than
 2850 **CKR_ENCRYPTED_DATA_LEN_RANGE**.
- 2851 • **CKR_ENCRYPTED_DATA_LEN_RANGE**: The ciphertext input to a decryption operation has been
 2852 determined to be invalid ciphertext solely on the basis of its length. Depending on the operation's
 2853 mechanism, this could mean that the ciphertext is too short, too long, or is not a multiple of some
 2854 particular block size. This return value has higher priority than **CKR_ENCRYPTED_DATA_INVALID**.
- 2855 • **CKR_EXCEEDED_MAX_ITERATIONS**: An iterative algorithm (for key pair generation, domain
 2856 parameter generation etc.) failed because we have exceeded the maximum number of iterations. This
 2857 error code has precedence over **CKR_FUNCTION_FAILED**. Examples of iterative algorithms include
 2858 DSA signature generation (retry if either $r = 0$ or $s = 0$) and generation of DSA primes p and q
 2859 specified in [FIPS PUB 186-4].
- 2860 • **CKR_FIPS_SELF_TEST_FAILED**: A FIPS 140-2 power-up self-test or conditional self-test failed. The
 2861 token entered an error state. Future calls to cryptographic functions on the token will return
 2862 **CKR_GENERAL_ERROR**. **CKR_FIPS_SELF_TEST_FAILED** has a higher precedence over
 2863 **CKR_GENERAL_ERROR**. This error may be returned by **C_Initialize**, if a power-up self-test failed,
 2864 by **C_GenerateRandom** or **C_SeedRandom**, if the continuous random number generator test failed,
 2865 or by **C_GenerateKeyPair**, if the pair-wise consistency test failed.
- 2866 • **CKR_FUNCTION_CANCELED**: The function was canceled in mid-execution. This happens to a
 2867 cryptographic function if the function makes a **CKN_SURRENDER** application callback which returns
 2868 **CKR_CANCEL** (see **CKR_CANCEL**). It also happens to a function that performs PIN entry through a
 2869 protected path. The method used to cancel a protected path PIN entry operation is device dependent.
- 2870 • **CKR_FUNCTION_NOT_PARALLEL**: There is currently no function executing in parallel in the
 2871 specified session. This is a legacy error code which is only returned by the legacy functions
 2872 **C_GetFunctionStatus** and **C_CancelFunction**.
- 2873 • **CKR_FUNCTION_NOT_SUPPORTED**: The requested function is not supported by this Cryptoki
 2874 library. Even unsupported functions in the Cryptoki API should have a "stub" in the library; this stub
 2875 should simply return the value **CKR_FUNCTION_NOT_SUPPORTED**.
- 2876 • **CKR_FUNCTION_REJECTED**: The signature request is rejected by the user.
- 2877 • **CKR_INFORMATION_SENSITIVE**: The information requested could not be obtained because the
 2878 token considers it sensitive, and is not able or willing to reveal it.
- 2879 • **CKR_KEY_CHANGED**: This value is only returned by **C_SetOperationState**. It indicates that one of
 2880 the keys specified is not the same key that was being used in the original saved session.
- 2881 • **CKR_KEY_FUNCTION_NOT_PERMITTED**: An attempt has been made to use a key for a
 2882 cryptographic purpose that the key's attributes are not set to allow it to do. For example, to use a key
 2883 for performing encryption, that key **MUST** have its **CKA_ENCRYPT** attribute set to **CK_TRUE** (the
 2884 fact that the key **MUST** have a **CKA_ENCRYPT** attribute implies that the key cannot be a private
 2885 key). This return value has lower priority than **CKR_KEY_TYPE_INCONSISTENT**.

- 2886 • **CKR_KEY_HANDLE_INVALID**: The specified key handle is not valid. It may be the case that the
2887 specified handle is a valid handle for an object which is not a key. We reiterate here that 0 is never a
2888 valid key handle.
- 2889 • **CKR_KEY_INDIGESTIBLE**: This error code can only be returned by **C_DigestKey**. It indicates that
2890 the value of the specified key cannot be digested for some reason (perhaps the key isn't a secret key,
2891 or perhaps the token simply can't digest this kind of key).
- 2892 • **CKR_KEY_NEEDED**: This value is only returned by **C_SetOperationState**. It indicates that the
2893 session state cannot be restored because **C_SetOperationState** needs to be supplied with one or
2894 more keys that were being used in the original saved session.
- 2895 • **CKR_KEY_NOT_NEEDED**: An extraneous key was supplied to **C_SetOperationState**. For example,
2896 an attempt was made to restore a session that had been performing a message digesting operation,
2897 and an encryption key was supplied.
- 2898 • **CKR_KEY_NOT_WRAPPABLE**: Although the specified private or secret key does not have its
2899 **CKA_EXTRACTABLE** attribute set to CK_FALSE, Cryptoki (or the token) is unable to wrap the key
2900 as requested (possibly the token can only wrap a given key with certain types of keys, and the
2901 wrapping key specified is not one of these types). Compare with **CKR_KEY_UNEXTRACTABLE**.
- 2902 • **CKR_KEY_SIZE_RANGE**: Although the requested keyed cryptographic operation could in principle
2903 be carried out, this Cryptoki library (or the token) is unable to actually do it because the supplied key's
2904 size is outside the range of key sizes that it can handle.
- 2905 • **CKR_KEY_TYPE_INCONSISTENT**: The specified key is not the correct type of key to use with the
2906 specified mechanism. This return value has a higher priority than
2907 **CKR_KEY_FUNCTION_NOT_PERMITTED**.
- 2908 • **CKR_KEY_UNEXTRACTABLE**: The specified private or secret key can't be wrapped because its
2909 **CKA_EXTRACTABLE** attribute is set to CK_FALSE. Compare with **CKR_KEY_NOT_WRAPPABLE**.
- 2910 • **CKR_LIBRARY_LOAD_FAILED**: The Cryptoki library could not load a dependent shared library.
- 2911 • **CKR_MECHANISM_INVALID**: An invalid mechanism was specified to the cryptographic operation.
2912 This error code is an appropriate return value if an unknown mechanism was specified or if the
2913 mechanism specified cannot be used in the selected token with the selected function.
- 2914 • **CKR_MECHANISM_PARAM_INVALID**: Invalid parameters were supplied to the mechanism specified
2915 to the cryptographic operation. Which parameter values are supported by a given mechanism can
2916 vary from token to token.
- 2917 • **CKR_NEED_TO_CREATE_THREADS**: This value can only be returned by **C_Initialize**. It is returned
2918 when two conditions hold:
 - 2919 1. The application called **C_Initialize** in a way which tells the Cryptoki library that application
2920 threads executing calls to the library cannot use native operating system methods to spawn new
2921 threads.
 - 2922 2. The library cannot function properly without being able to spawn new threads in the above
2923 fashion.
- 2924 • **CKR_NO_EVENT**: This value can only be returned by **C_WaitForSlotEvent**. It is returned when
2925 **C_WaitForSlotEvent** is called in non-blocking mode and there are no new slot events to return.
- 2926 • **CKR_OBJECT_HANDLE_INVALID**: The specified object handle is not valid. We reiterate here that 0
2927 is never a valid object handle.
- 2928 • **CKR_OPERATION_ACTIVE**: There is already an active operation (or combination of active
2929 operations) which prevents Cryptoki from activating the specified operation. For example, an active
2930 object-searching operation would prevent Cryptoki from activating an encryption operation with
2931 **C_EncryptInit**. Or, an active digesting operation and an active encryption operation would prevent
2932 Cryptoki from activating a signature operation. Or, on a token which doesn't support simultaneous
2933 dual cryptographic operations in a session (see the description of the
2934 **CKF_DUAL_CRYPTO_OPERATIONS** flag in the **CK_TOKEN_INFO** structure), an active signature
2935 operation would prevent Cryptoki from activating an encryption operation.

- 2936 • CKR_OPERATION_NOT_INITIALIZED: There is no active operation of an appropriate type in the
2937 specified session. For example, an application cannot call **C_Encrypt** in a session without having
2938 called **C_EncryptInit** first to activate an encryption operation.
- 2939 • CKR_PARAMETER_SET_NOT_SUPPORTED: This parameter set is not supported by this token.
2940 Used with XMSS, XMSS^{MT}, ML-KEM, ML-DSA and SLH-DSA mechanisms.
- 2941 • CKR_PENDING: This value is returned if the operation is running asynchronously.
- 2942 • CKR_PIN_EXPIRED: The specified PIN has expired, and the requested operation cannot be carried
2943 out unless **C_SetPIN** is called to change the PIN value. Whether or not the normal user's PIN on a
2944 token ever expires varies from token to token.
- 2945 • CKR_PIN_INCORRECT: The specified PIN is incorrect, *i.e.*, does not match the PIN stored on the
2946 token. More generally-- when authentication to the token involves something other than a PIN-- the
2947 attempt to authenticate the user has failed.
- 2948 • CKR_PIN_INVALID: The specified PIN has invalid characters in it. This return code only applies to
2949 functions which attempt to set a PIN.
- 2950 • CKR_PIN_LEN_RANGE: The specified PIN is too long or too short. This return code only applies to
2951 functions which attempt to set a PIN.
- 2952 • CKR_PIN_LOCKED: The specified PIN is "locked", and cannot be used. That is, because some
2953 particular number of failed authentication attempts has been reached, the token is unwilling to permit
2954 further attempts at authentication. Depending on the token, the specified PIN may or may not remain
2955 locked indefinitely.
- 2956 • CKR_PIN_TOO_WEAK: The specified PIN is too weak so that it could be easy to guess. If the PIN is
2957 too short, CKR_PIN_LEN_RANGE should be returned instead. This return code only applies to
2958 functions which attempt to set a PIN.
- 2959 • CKR_PUBLIC_KEY_INVALID: The public key fails a public key validation. For example, an EC public
2960 key fails the public key validation specified in Section 5.2.2 of [ANSI X9.62]. This error code may be
2961 returned by **C_CreateObject**, when the public key is created, or by **C_VerifyInit**,
2962 **C_VerifySignatureInit** or **C_VerifyRecoverInit**, when the public key is used. It may also be returned
2963 by **C_DeriveKey**, in preference to CKR_MECHANISM_PARAM_INVALID, if the other party's public
2964 key specified in the mechanism's parameters is invalid.
- 2965 • CKR_RANDOM_NO_RNG: This value can be returned by **C_SeedRandom** and
2966 **C_GenerateRandom**. It indicates that the specified token doesn't have a random number generator.
2967 This return value has higher priority than CKR_RANDOM_SEED_NOT_SUPPORTED.
- 2968 • CKR_RANDOM_SEED_NOT_SUPPORTED: This value can only be returned by **C_SeedRandom**. It
2969 indicates that the token's random number generator does not accept seeding from an application.
2970 This return value has lower priority than CKR_RANDOM_NO_RNG.
- 2971 • CKR_SAVED_STATE_INVALID: This value can only be returned by **C_SetOperationState**. It
2972 indicates that the supplied saved cryptographic operations state is invalid, and so it cannot be
2973 restored to the specified session.
- 2974 • CKR_SEED_RANDOM_REQUIRED: This value can only be returned by **C_GenerateRandom**. It
2975 indicates that the token's random number generator has not yet been seeded, or requires re-seeding,
2976 by **C_SeedRandom**.
- 2977 • CKR_SESSION_ASYNC_NOT_SUPPORTED: This value is returned if the token doesn't support
2978 async operations.
- 2979 • CKR_SESSION_COUNT: This value can only be returned by **C_OpenSession**. It indicates that the
2980 attempt to open a session failed, either because the token has too many sessions already open, or
2981 because the token has too many read/write sessions already open.
- 2982 • CKR_SESSION_EXISTS: This value can only be returned by **C_InitToken**. It indicates that a session
2983 with the token is already open, and so the token cannot be initialized.
- 2984 • CKR_SESSION_PARALLEL_NOT_SUPPORTED: The specified token does not support parallel
2985 sessions. This is a legacy error code—in Cryptoki Version 2.01 and up, *no* token supports parallel
2986 sessions. CKR_SESSION_PARALLEL_NOT_SUPPORTED can only be returned by

2987 **C_OpenSession**, and it is only returned when **C_OpenSession** is called in a particular [deprecated]
 2988 way.

- 2989 • CKR_SESSION_READ_ONLY: The specified session was unable to accomplish the desired action
 2990 because it is a read-only session. This return value has lower priority than
 2991 CKR_TOKEN_WRITE_PROTECTED.
- 2992 • CKR_SESSION_READ_ONLY_EXISTS: A read-only session already exists, and so the SO cannot
 2993 be logged in.
- 2994 • CKR_SESSION_READ_WRITE_SO_EXISTS: A read/write SO session already exists, and so a
 2995 read-only session cannot be opened.
- 2996 • CKR_SIGNATURE_LEN_RANGE: The provided signature/MAC can be seen to be invalid solely on
 2997 the basis of its length. This return value has higher priority than CKR_SIGNATURE_INVALID.
- 2998 • CKR_SIGNATURE_INVALID: The provided signature/MAC is invalid. This return value has lower
 2999 priority than CKR_SIGNATURE_LEN_RANGE.
- 3000 • CKR_SLOT_ID_INVALID: The specified slot ID is not valid.
- 3001 • CKR_STATE_UNSAVEABLE: The cryptographic operations state of the specified session cannot be
 3002 saved for some reason (possibly the token is simply unable to save the current state). This return
 3003 value has lower priority than CKR_OPERATION_NOT_INITIALIZED.
- 3004 • CKR_TEMPLATE_INCOMPLETE: The template specified for creating an object is incomplete, and
 3005 lacks some necessary attributes. See Section 4.1 for more information.
- 3006 • CKR_TEMPLATE_INCONSISTENT: The template specified for creating an object has conflicting
 3007 attributes. See Section 4.1 for more information.
- 3008 • CKR_TOKEN_NOT_RECOGNIZED: The Cryptoki library and/or slot does not recognize the token in
 3009 the slot.
- 3010 • CKR_TOKEN_WRITE_PROTECTED: The requested action could not be performed because the
 3011 token is write-protected. This return value has higher priority than CKR_SESSION_READ_ONLY.
- 3012 • CKR_UNWRAPPING_KEY_HANDLE_INVALID: This value can only be returned by **C_UnwrapKey**.
 3013 It indicates that the key handle specified to be used to unwrap another key is not valid.
- 3014 • CKR_UNWRAPPING_KEY_SIZE_RANGE: This value can only be returned by **C_UnwrapKey**. It
 3015 indicates that although the requested unwrapping operation could in principle be carried out, this
 3016 Cryptoki library (or the token) is unable to actually do it because the supplied key's size is outside the
 3017 range of key sizes that it can handle.
- 3018 • CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT: This value can only be returned by
 3019 **C_UnwrapKey**. It indicates that the type of the key specified to unwrap another key is not consistent
 3020 with the mechanism specified for unwrapping.
- 3021 • CKR_USER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It indicates that
 3022 the specified user cannot be logged into the session, because it is already logged into the session.
 3023 For example, if an application has an open SO session, and it attempts to log the SO into it, it will
 3024 receive this error code.
- 3025 • CKR_USER_ANOTHER_ALREADY_LOGGED_IN: This value can only be returned by **C_Login**. It
 3026 indicates that the specified user cannot be logged into the session, because another user is already
 3027 logged into the session. For example, if an application has an open SO session, and it attempts to log
 3028 the normal user into it, it will receive this error code.
- 3029 • CKR_USER_NOT_LOGGED_IN: The desired action cannot be performed because the appropriate
 3030 user (or *an* appropriate user) is not logged in. One example is that a session cannot be logged out
 3031 unless it is logged in. Another example is that a private object cannot be created on a token unless
 3032 the session attempting to create it is logged in as the normal user. A final example is that
 3033 cryptographic operations on certain tokens cannot be performed unless the normal user is logged in.
- 3034 • CKR_USER_PIN_NOT_INITIALIZED: This value can only be returned by **C_Login**. It indicates that
 3035 the normal user's PIN has not yet been initialized with **C_InitPIN**.

- 3036 • **CKR_USER_TOO_MANY_TYPES**: An attempt was made to have more distinct users simultaneously
 3037 logged into the token than the token and/or library permits. For example, if some application has an
 3038 open SO session, and another application attempts to log the normal user into a session, the attempt
 3039 may return this error. It is not required to, however. Only if the simultaneous distinct users cannot be
 3040 supported does **C_Login** have to return this value. Note that this error code generalizes to true multi-
 3041 user tokens.
- 3042 • **CKR_USER_TYPE_INVALID**: An invalid value was specified as a **CK_USER_TYPE**. Valid types are
 3043 **CKU_SO**, **CKU_USER**, and **CKU_CONTEXT_SPECIFIC**.
- 3044 • **CKR_WRAPPED_KEY_INVALID**: This value can only be returned by **C_UnwrapKey**. It indicates that
 3045 the provided wrapped key is not valid. If a call is made to **C_UnwrapKey** to unwrap a particular type
 3046 of key (*i.e.*, some particular key type is specified in the template provided to **C_UnwrapKey**), and the
 3047 wrapped key provided to **C_UnwrapKey** is recognizably not a wrapped key of the proper type, then
 3048 **C_UnwrapKey** should return **CKR_WRAPPED_KEY_INVALID**. This return value has lower priority
 3049 than **CKR_WRAPPED_KEY_LEN_RANGE**.
- 3050 • **CKR_WRAPPED_KEY_LEN_RANGE**: This value can only be returned by **C_UnwrapKey**. It
 3051 indicates that the provided wrapped key can be seen to be invalid solely on the basis of its length.
 3052 This return value has higher priority than **CKR_WRAPPED_KEY_INVALID**.
- 3053 • **CKR_WRAPPING_KEY_HANDLE_INVALID**: This value can only be returned by **C_WrapKey**. It
 3054 indicates that the key handle specified to be used to wrap another key is not valid.
- 3055 • **CKR_WRAPPING_KEY_SIZE_RANGE**: This value can only be returned by **C_WrapKey**. It indicates
 3056 that although the requested wrapping operation could in principle be carried out, this Cryptoki library
 3057 (or the token) is unable to actually do it because the supplied wrapping key's size is outside the range
 3058 of key sizes that it can handle.
- 3059 • **CKR_WRAPPING_KEY_TYPE_INCONSISTENT**: This value can only be returned by **C_WrapKey**. It
 3060 indicates that the type of the key specified to wrap another key is not consistent with the mechanism
 3061 specified for wrapping.
- 3062 • **CKR_OPERATION_CANCEL_FAILED**: This value can only be returned by **C_SessionCancel**. It
 3063 means that one or more of the requested operations could not be cancelled for implementation or
 3064 vendor-specific reasons.

3065 5.1.7 More on relative priorities of Cryptoki errors

3066 In general, when a Cryptoki call is made, error codes from Section 5.1.1 (other than **CKR_OK**) take
 3067 precedence over error codes from Section 5.1.2, which take precedence over error codes from Section
 3068 5.1.3, which take precedence over error codes from Section 5.1.6. One minor implication of this is that
 3069 functions that use a session handle (*i.e.*, *most* functions!) never return the error code
 3070 **CKR_TOKEN_NOT_PRESENT** (they return **CKR_SESSION_HANDLE_INVALID** instead). Other than
 3071 these precedences, if more than one error code applies to the result of a Cryptoki call, any of the
 3072 applicable error codes may be returned. Exceptions to this rule will be explicitly mentioned in the
 3073 descriptions of functions.

3074 5.1.8 Error code “gotchas”

3075 Here is a short list of a few particular things about return values that Cryptoki developers might want to be
 3076 aware of:

- 3077 1. As mentioned in Sections 5.1.2 and 5.1.3, a Cryptoki library may not be able to make a distinction
 3078 between a token being removed *before* a function invocation and a token being removed *during* a
 3079 function invocation.
- 3080 2. As mentioned in Section 5.1.2, an application should never count on getting a
 3081 **CKR_SESSION_CLOSED** error.
- 3082 3. The difference between **CKR_DATA_INVALID** and **CKR_DATA_LEN_RANGE** can be somewhat
 3083 subtle. Unless an application *needs* to be able to distinguish between these return values, it is best to
 3084 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 with PKCS padding. Suppose that, initially, 8 bytes of ciphertext are passed to the **C_DecryptUpdate** function. The block size of DES is 8 bytes, but the PKCS padding makes it unclear at this stage whether the ciphertext was produced from encrypting a 0-byte string, or from encrypting some string of length at least 8 bytes. Hence the call to **C_DecryptUpdate** should return 0 bytes of plaintext. If a single additional byte of ciphertext is supplied by a subsequent call to **C_DecryptUpdate**, then that call should return 8 bytes of plaintext (one full DES block).

5.3 Disclaimer concerning sample code

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

5.4 General-purpose functions

Cryptoki provides the following general-purpose functions:

5.4.1 C_Initialize

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

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

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

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

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

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

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

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

C_Initialize should be the first Cryptoki call made by an application, except for calls to **C_GetFunctionList**, **C_GetInterfaceList**, or **C_GetInterface**. What this function actually does is

3183 implementation-dependent; typically, it might cause Cryptoki to initialize its internal memory buffers, or
3184 any other resources it requires.

3185 If several applications are using Cryptoki, each one should call **C_Initialize**. Every call to **C_Initialize**
3186 should (eventually) be succeeded by a single call to **C_Finalize**. See [PKCS11-UG] for further details.

3187 Return values: CKR_ARGUMENTS_BAD, CKR_CANT_LOCK,
3188 CKR_CRYPTOKI_ALREADY_INITIALIZED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3189 CKR_HOST_MEMORY, CKR_NEED_TO_CREATE_THREADS, CKR_OK.

3190 Example: see **C_GetInfo**.

3191 5.4.2 C_Finalize

```
3192 CK_DECLARE_FUNCTION(CK_RV, C_Finalize) (  
3193     CK_VOID_PTR pReserved  
3194 );
```

3195 **C_Finalize** is called to indicate that an application is finished with the Cryptoki library. It should be the last
3196 Cryptoki call made by an application. The *pReserved* parameter is reserved for future versions; for this
3197 version, it should be set to NULL_PTR (if **C_Finalize** is called with a non-NULL_PTR value for
3198 *pReserved*, it should return the value **CKR_ARGUMENTS_BAD**).

3199 If several applications are using Cryptoki, each one should call **C_Finalize**. Each application's call to
3200 **C_Finalize** should be preceded by a single call to **C_Initialize**; in between the two calls, an application
3201 can make calls to other Cryptoki functions. See [PKCS11-UG] for further details.

3202 *Despite the fact that the parameters supplied to C_Initialize can in general allow for safe multi-threaded*
3203 *access to a Cryptoki library, the behavior of C_Finalize is nevertheless undefined if it is called by an*
3204 *application while other threads of the application are making Cryptoki calls. The exception to this*
3205 *exceptional behavior of C_Finalize occurs when a thread calls C_Finalize while another of the*
3206 *application's threads is blocking on Cryptoki's C_WaitForSlotEvent function. When this happens, the*
3207 *blocked thread becomes unblocked and returns the value CKR_CRYPTOKI_NOT_INITIALIZED. See*
3208 *C_WaitForSlotEvent for more information.*

3209 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3210 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

3211 Example: see **C_GetInfo**.

3212 5.4.3 C_GetInfo

```
3213 CK_DECLARE_FUNCTION(CK_RV, C_GetInfo) (  
3214     CK_INFO_PTR pInfo  
3215 );
```

3216 **C_GetInfo** returns general information about Cryptoki. *pInfo* points to the location that receives the
3217 information.

3218 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
3219 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

3220 Example:

```
3221 CK_INFO info;  
3222 CK_RV rv;  
3223 CK_C_INITIALIZE_ARGS InitArgs;  
3224  
3225 InitArgs.CreateMutex = &MyCreateMutex;  
3226 InitArgs.DestroyMutex = &MyDestroyMutex;  
3227 InitArgs.LockMutex = &MyLockMutex;
```

```

3228 InitArgs.UnlockMutex = &MyUnlockMutex;
3229 InitArgs.flags = CKF_OS_LOCKING_OK;
3230 InitArgs.pReserved = NULL_PTR;
3231
3232 rv = C_Initialize((CK_VOID_PTR)&InitArgs);
3233 assert(rv == CKR_OK);
3234
3235 rv = C_GetInfo(&info);
3236 assert(rv == CKR_OK);
3237 if(info.cryptokiVersion.major == 2) {
3238     /* Do lots of interesting cryptographic things with the token */
3239     .
3240     .
3241 }
3242
3243 rv = C_Finalize(NULL_PTR);
3244 assert(rv == CKR_OK);

```

3245 5.4.4 C_GetFunctionList

```

3246 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionList)(
3247     CK_FUNCTION_LIST_PTR_PTR ppFunctionList
3248 );

```

3249 **C_GetFunctionList** obtains a pointer to the Cryptoki library's list of function pointers. *ppFunctionList*
3250 points to a value which will receive a pointer to the library's **CK_FUNCTION_LIST** structure, which in turn
3251 contains function pointers for all the Cryptoki API routines in the library. *The pointer thus obtained may*
3252 *point into memory which is owned by the Cryptoki library, and which may or may not be writable.* Whether
3253 or not this is the case, no attempt should be made to write to this memory.

3254 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
3255 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
3256 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

3257 Return values: CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3258 CKR_HOST_MEMORY, CKR_OK.

3259 Example:

```

3260 CK_FUNCTION_LIST_PTR pFunctionList;
3261 CK_C_Initialize pC_Initialize;
3262 CK_RV rv;
3263
3264 /* It's OK to call C_GetFunctionList before calling C_Initialize */
3265 rv = C_GetFunctionList(&pFunctionList);
3266 assert(rv == CKR_OK);
3267 pC_Initialize = pFunctionList -> C_Initialize;
3268
3269 /* Call the C_Initialize function in the library */
3270 rv = (*pC_Initialize)(NULL_PTR);

```

5.4.5 C_GetInterfaceList

```
CK_DECLARE_FUNCTION(CK_RV, C_GetInterfaceList) (
    CK_INTERFACE_PTR      pInterfaceList,
    CK_ULONG_PTR          pulCount
);
```

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

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

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

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

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

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

Return values: **CKR_BUFFER_TOO_SMALL**, **CKR_ARGUMENTS_BAD**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**.

Example:

```
CK_ULONG ulCount=0;
CK_INTERFACE_PTR interfaceList=NULL;
CK_RV rv;
int I;

/* get number of interfaces */
rv = C_GetInterfaceList(NULL,&ulCount);
if (rv == CKR_OK) {
    /* get copy of interfaces */
    interfaceList = (CK_INTERFACE_PTR)malloc(ulCount*sizeof(CK_INTERFACE));
    rv = C_GetInterfaceList(interfaceList,&ulCount);
    for(i=0;i<ulCount;i++) {
        printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
            interfaceList[i].pInterfaceName,
            ((CK_VERSION *)interfaceList[i].pFunctionList)->major,
            ((CK_VERSION *)interfaceList[i].pFunctionList)->minor,
            interfaceList[i].pFunctionList,
```

```

3317         interfaceList[i].flags);
3318     }
3319 }
3320

```

3321 5.4.6 C_GetInterface

```

3322 CK_DECLARE_FUNCTION(CK_RV,C_GetInterface) (
3323     CK_UTF8CHAR_PTR      pInterfaceName,
3324     CK_VERSION_PTR       pVersion,
3325     CK_INTERFACE_PTR_PTR ppInterface,
3326     CK_FLAGS              flags
3327 );

```

3328 **C_GetInterface** is used to obtain an interface supported by a Cryptoki library. *pInterfaceName* specifies
3329 the name of the interface, *pVersion* specifies the interface version, *ppInterface* points to the location that
3330 receives the interface, *flags* specifies the required interface flags.

3331 There are multiple ways for an application to specify a particular interface when calling **C_GetInterface**:

- 3332 1. If *pInterfaceName* is not NULL_PTR, the name of the interface returned must match. If
3333 *pInterfaceName* is NULL_PTR, the cryptoki library can return a default interface of its choice
- 3334 2. If *pVersion* is not NULL_PTR, the version of the interface returned must match. If *pVersion* is
3335 NULL_PTR, the cryptoki library can return an interface of any version
- 3336 3. If *flags* is non-zero, the interface returned must match all of the supplied flag values (but may include
3337 additional flags not specified). If *flags* is 0, the cryptoki library can return an interface with any flags

3338 **C_GetInterface** obtains (in **pFunctionList* of each interface) a pointer to the Cryptoki library's list of
3339 function pointers. *The pointer thus obtained may point into memory which is owned by the Cryptoki*
3340 *library, and which may or may not be writable.* Whether or not this is the case, no attempt should be
3341 made to write to this memory. The same caveat applies to the interface names returned.

3342 **C_GetFunctionList**, **C_GetInterfaceList**, and **C_GetInterface** are the only Cryptoki functions which an
3343 application may call before calling **C_Initialize**. It is provided to make it easier and faster for applications
3344 to use shared Cryptoki libraries and to use more than one Cryptoki library simultaneously.

3345 Return values: CKR_BUFFER_TOO_SMALL, CKR_ARGUMENTS_BAD, CKR_FUNCTION_FAILED,
3346 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK.

3347 Example:

```

3348 CK_INTERFACE_PTR interface;
3349 CK_RV rv;
3350 CK_VERSION version;
3351 CK_FLAGS flags=CKF_INTERFACE_FORK_SAFE;
3352
3353 /* get default interface */
3354 rv = C_GetInterface(NULL,NULL,&interface,flags);
3355 if (rv == CKR_OK) {
3356     printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
3357         interface->pInterfaceName,
3358         ((CK_VERSION *)interface->pFunctionList)->major,
3359         ((CK_VERSION *)interface->pFunctionList)->minor,
3360         interface->pFunctionList,

```

```

3361         interface->flags);
3362     }
3363
3364     /* get default standard interface */
3365     rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",NULL,&interface,flags);
3366     if (rv == CKR_OK) {
3367         printf("interface %s version %d.%d funcs %p flags 0x%lu\n",
3368             interface->pInterfaceName,
3369             ((CK_VERSION *)interface->pFunctionList)->major,
3370             ((CK_VERSION *)interface->pFunctionList)->minor,
3371             interface->pFunctionList,
3372             interface->flags);
3373     }
3374
3375     /* get specific standard version interface */
3376     version.major=3;
3377     version.minor=0;
3378     rv = C_GetInterface((CK_UTF8CHAR_PTR)"PKCS 11",&version,&interface,flags);
3379     if (rv == CKR_OK) {
3380         CK_FUNCTION_LIST_3_0_PTR pkcs11=interface->pFunctionList;
3381
3382         /* ... use the new functions */
3383         pkcs11->C_LoginUser(hSession,userType,pPin,ulPinLen,
3384                             pUsername,ulUsernameLen);
3385     }
3386
3387     /* get specific vendor version interface */
3388     version.major=1;
3389     version.minor=0;
3390     rv = C_GetInterface((CK_UTF8CHAR_PTR)
3391         "Vendor VendorName",&version,&interface,flags);
3392     if (rv == CKR_OK) {
3393         CK_FUNCTION_LIST_VENDOR_1_0_PTR pkcs11=interface->pFunctionList;
3394
3395         /* ... use vendor specific functions */
3396         pkcs11->C_VendorFunction1(param1,param2,param3);
3397     }
3398

```

3399 5.5 Slot and token management functions

3400 Cryptoki provides the following functions for slot and token management:

5.5.1 C_GetSlotList

```
CK_DECLARE_FUNCTION(CK_RV, C_GetSlotList)(
    CK_BBOOL tokenPresent,
    CK_SLOT_ID_PTR pSlotList,
    CK_ULONG_PTR pulCount
);
```

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

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

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

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

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

Return values: **CKR_ARGUMENTS_BAD**, **CKR_BUFFER_TOO_SMALL**, **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**.

Example:

```
CK_ULONG ulSlotCount, ulSlotWithTokenCount;
CK_SLOT_ID_PTR pSlotList, pSlotWithTokenList;
CK_RV rv;

/* Get list of all slots */
rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulSlotCount);
if (rv == CKR_OK) {
    pSlotList =
        (CK_SLOT_ID_PTR) malloc(ulSlotCount*sizeof(CK_SLOT_ID));
    rv = C_GetSlotList(CK_FALSE, pSlotList, &ulSlotCount);
    if (rv == CKR_OK) {
```

```

3448     /* Now use that list of all slots */
3449     .
3450     .
3451 }
3452
3453     free(pSlotList);
3454 }
3455
3456 /* Get list of all slots with a token present */
3457 pSlotWithTokenList = (CK_SLOT_ID_PTR) malloc(0);
3458 ulSlotWithTokenCount = 0;
3459 while (1) {
3460     rv = C_GetSlotList(
3461         CK_TRUE, pSlotWithTokenList, &ulSlotWithTokenCount);
3462     if (rv != CKR_BUFFER_TOO_SMALL)
3463         break;
3464     pSlotWithTokenList = realloc(
3465         pSlotWithTokenList,
3466         ulSlotWithTokenList*sizeof(CK_SLOT_ID));
3467 }
3468
3469 if (rv == CKR_OK) {
3470     /* Now use that list of all slots with a token present */
3471     .
3472     .
3473 }
3474
3475 free(pSlotWithTokenList);

```

5.5.2 C_GetSlotInfo

```

3477 CK_DECLARE_FUNCTION(CK_RV, C_GetSlotInfo)(
3478     CK_SLOT_ID slotID,
3479     CK_SLOT_INFO_PTR pInfo
3480 );

```

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

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

Example: see **C_GetTokenInfo**.

5.5.3 C_GetTokenInfo

```

3488 CK_DECLARE_FUNCTION(CK_RV, C_GetTokenInfo)(

```

```

3489     CK_SLOT_ID slotID,
3490     CK_TOKEN_INFO_PTR pInfo
3491 );

```

3492 **C_GetTokenInfo** obtains information about a particular token in the system. *slotID* is the ID of the token's
3493 slot; *pInfo* points to the location that receives the token information.

3494 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3495 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3496 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT,
3497 CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

3498 **Example:**

```

3499 CK_ULONG ulCount;
3500 CK_SLOT_ID_PTR pSlotList;
3501 CK_SLOT_INFO slotInfo;
3502 CK_TOKEN_INFO tokenInfo;
3503 CK_RV rv;
3504
3505 rv = C_GetSlotList(CK_FALSE, NULL_PTR, &ulCount);
3506 if ((rv == CKR_OK) && (ulCount > 0)) {
3507     pSlotList = (CK_SLOT_ID_PTR) malloc(ulCount*sizeof(CK_SLOT_ID));
3508     rv = C_GetSlotList(CK_FALSE, pSlotList, &ulCount);
3509     assert(rv == CKR_OK);
3510
3511     /* Get slot information for first slot */
3512     rv = C_GetSlotInfo(pSlotList[0], &slotInfo);
3513     assert(rv == CKR_OK);
3514
3515     /* Get token information for first slot */
3516     rv = C_GetTokenInfo(pSlotList[0], &tokenInfo);
3517     if (rv == CKR_TOKEN_NOT_PRESENT) {
3518         .
3519         .
3520     }
3521     .
3522     .
3523     free(pSlotList);
3524 }

```

3525 5.5.4 C_WaitForSlotEvent

```

3526 CK_DECLARE_FUNCTION(CK_RV, C_WaitForSlotEvent)(
3527     CK_FLAGS flags,
3528     CK_SLOT_ID_PTR pSlot,
3529     CK_VOID_PTR pReserved
3530 );

```

3531 **C_WaitForSlotEvent** waits for a slot event, such as token insertion or token removal, to occur. *flags*
3532 determines whether or not the **C_WaitForSlotEvent** call blocks (*i.e.*, waits for a slot event to occur); *pSlot*
3533 points to a location which will receive the ID of the slot that the event occurred in. *pReserved* is reserved
3534 for future versions; for this version of Cryptoki, it should be `NULL_PTR`.

3535 At present, the only flag defined for use in the *flags* argument is **CKF_DONT_BLOCK**:

3536 Internally, each Cryptoki application has a flag for each slot which is used to track whether or not any
3537 unrecognized events involving that slot have occurred. When an application initially calls **C_Initialize**,
3538 every slot's event flag is cleared. Whenever a slot event occurs, the flag corresponding to the slot in
3539 which the event occurred is set.

3540 If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and some
3541 slot's event flag is set, then that event flag is cleared, and the call returns with the ID of that slot in the
3542 location pointed to by *pSlot*. If more than one slot's event flag is set at the time of the call, one such slot is
3543 chosen by the library to have its event flag cleared and to have its slot ID returned.

3544 If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag set in the *flags* argument, and no
3545 slot's event flag is set, then the call returns with the value **CKR_NO_EVENT**. In this case, the contents of
3546 the location pointed to by *pSlot* when **C_WaitForSlotEvent** are undefined.

3547 If **C_WaitForSlotEvent** is called with the **CKF_DONT_BLOCK** flag clear in the *flags* argument, then the
3548 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,
3549 **C_WaitForSlotEvent** will wait until some slot's event flag becomes set. If a thread of an application has a
3550 **C_WaitForSlotEvent** call blocking when another thread of that application calls **C_Finalize**, the
3551 **C_WaitForSlotEvent** call returns with the value **CKR_CRYPTOKI_NOT_INITIALIZED**.

3552 *Although the parameters supplied to C_Initialize can in general allow for safe multi-threaded access to a*
3553 *Cryptoki library, C_WaitForSlotEvent is exceptional in that the behavior of Cryptoki is undefined if*
3554 *multiple threads of a single application make simultaneous calls to C_WaitForSlotEvent.*

3555 Return values: **CKR_ARGUMENTS_BAD**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
3556 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_NO_EVENT**,
3557 **CKR_OK**.

3558 Example:

```
3559 CK_FLAGS flags = 0;  
3560 CK_SLOT_ID slotID;  
3561 CK_SLOT_INFO slotInfo;  
3562 CK_RV rv;  
3563 .  
3564 .  
3565 /* Block and wait for a slot event */  
3566 rv = C_WaitForSlotEvent(flags, &slotID, NULL_PTR);  
3567 assert(rv == CKR_OK);  
3568  
3569 /* See what's up with that slot */  
3570 rv = C_GetSlotInfo(slotID, &slotInfo);  
3571 assert(rv == CKR_OK);  
3572
```

3573 5.5.5 C_GetMechanismList

```
3574 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismList)(  
3575     CK_SLOT_ID slotID,  
3576     CK_MECHANISM_TYPE_PTR pMechanismList,
```

```

3577         CK_ULONG_PTR pulCount
3578     );

```

3579 **C_GetMechanismList** is used to obtain a list of mechanism types supported by a token. *SlotID* is the ID
3580 of the token's slot; *pulCount* points to the location that receives the number of mechanisms.

3581 There are two ways for an application to call **C_GetMechanismList**:

- 3582 1. If *pMechanismList* is **NULL_PTR**, then all that **C_GetMechanismList** does is return (in **pulCount*)
3583 the number of mechanisms, without actually returning a list of mechanisms. The contents of
3584 **pulCount* on entry to **C_GetMechanismList** has no meaning in this case, and the call returns the
3585 value **CKR_OK**.
- 3586 2. If *pMechanismList* is not **NULL_PTR**, then **pulCount* MUST contain the size (in terms of
3587 **CK_MECHANISM_TYPE** elements) of the buffer pointed to by *pMechanismList*. If that buffer is large
3588 enough to hold the list of mechanisms, then the list is returned in it, and **CKR_OK** is returned. If not,
3589 then the call to **C_GetMechanismList** returns the value **CKR_BUFFER_TOO_SMALL**. In either
3590 case, the value **pulCount* is set to hold the number of mechanisms.

3591 Because **C_GetMechanismList** does not allocate any space of its own, an application will often call
3592 **C_GetMechanismList** twice. However, this behavior is by no means required.

3593 Return values: **CKR_BUFFER_TOO_SMALL**, **CKR_CRYPTOKI_NOT_INITIALIZED**,
3594 **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**, **CKR_DEVICE_REMOVED**,
3595 **CKR_FUNCTION_FAILED**, **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**,
3596 **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
3597 **CKR_ARGUMENTS_BAD**.

3598 Example:

```

3599 CK_SLOT_ID slotID;
3600 CK_ULONG ulCount;
3601 CK_MECHANISM_TYPE_PTR pMechanismList;
3602 CK_RV rv;
3603
3604 .
3605 .
3606 rv = C_GetMechanismList(slotID, NULL_PTR, &ulCount);
3607 if ((rv == CKR_OK) && (ulCount > 0)) {
3608     pMechanismList =
3609         (CK_MECHANISM_TYPE_PTR)
3610         malloc(ulCount*sizeof(CK_MECHANISM_TYPE));
3611     rv = C_GetMechanismList(slotID, pMechanismList, &ulCount);
3612     if (rv == CKR_OK) {
3613         .
3614         .
3615     }
3616     free(pMechanismList);
3617 }

```

3618 5.5.6 C_GetMechanismInfo

```

3619 CK_DECLARE_FUNCTION(CK_RV, C_GetMechanismInfo) (
3620     CK_SLOT_ID slotID,
3621     CK_MECHANISM_TYPE type,

```

```

3622         CK_MECHANISM_INFO_PTR pInfo
3623     );

```

3624 **C_GetMechanismInfo** obtains information about a particular mechanism possibly supported by a token. *slotID* is the ID of the token's slot; *type* is the type of mechanism; *pInfo* points to the location that receives the mechanism information.

3627 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_MECHANISM_INVALID, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT, CKR_TOKEN_NOT_RECOGNIZED, CKR_ARGUMENTS_BAD.

3631 **Example:**

```

3632     CK_SLOT_ID slotID;
3633     CK_MECHANISM_INFO info;
3634     CK_RV rv;
3635
3636     .
3637     .
3638     /* Get information about the CKM_MD2 mechanism for this token */
3639     rv = C_GetMechanismInfo(slotID, CKM_MD2, &info);
3640     if (rv == CKR_OK) {
3641         if (info.flags & CKF_DIGEST) {
3642             .
3643             .
3644         }
3645     }

```

3646 5.5.7 C_InitToken

```

3647     CK_DECLARE_FUNCTION(CK_RV, C_InitToken) (
3648         CK_SLOT_ID slotID,
3649         CK_UTF8CHAR_PTR pPin,
3650         CK_ULONG ulPinLen,
3651         CK_UTF8CHAR_PTR pLabel
3652     );

```

3653 **C_InitToken** initializes a token. *slotID* is the ID of the token's slot; *pPin* points to the SO's initial PIN (which need *not* be null-terminated); *ulPinLen* is the length in bytes of the PIN; *pLabel* points to the 32-byte label of the token (which **MUST** be padded with blank characters, and which **MUST** *not* be null-terminated). This standard allows PIN values to contain any valid UTF8 character, but the token may impose subset restrictions.

3658 If the token has not been initialized (i.e. new from the factory), then the *pPin* parameter becomes the initial value of the SO PIN. If the token is being reinitialized, the *pPin* parameter is checked against the existing SO PIN to authorize the initialization operation. In both cases, the SO PIN is the value *pPin* after the function completes successfully. If the SO PIN is lost, then the card **MUST** be reinitialized using a mechanism outside the scope of this standard. The **CKF_TOKEN_INITIALIZED** flag in the **CK_TOKEN_INFO** structure indicates the action that will result from calling **C_InitToken**. If set, the token will be reinitialized, and the client **MUST** supply the existing SO password in *pPin*.

3665 When a token is initialized, all objects that can be destroyed are destroyed (i.e., all except for "indestructible" objects such as keys built into the token). Also, access by the normal user is disabled until

3667 the SO sets the normal user's PIN. Depending on the token, some "default" objects may be created, and
3668 attributes of some objects may be set to default values.

3669 If the token has a "protected authentication path", as indicated by the
3670 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
3671 that there is some way for a user to be authenticated to the token without having the application send a
3672 PIN through the Cryptoki library. One such possibility is that the user enters a PIN on a PINpad on the
3673 token itself, or on the slot device. To initialize a token with such a protected authentication path, the *pPin*
3674 parameter to **C_InitToken** should be **NULL_PTR**. During the execution of **C_InitToken**, the SO's PIN will
3675 be entered through the protected authentication path.

3676 If the token has a protected authentication path other than a PINpad, then it is token-dependent whether
3677 or not **C_InitToken** can be used to initialize the token.

3678 A token cannot be initialized if Cryptoki detects that *any* application has an open session with it; when a
3679 call to **C_InitToken** is made under such circumstances, the call fails with error **CKR_SESSION_EXISTS**.
3680 Unfortunately, it may happen when **C_InitToken** is called that some other application *does* have an open
3681 session with the token, but Cryptoki cannot detect this, because it cannot detect anything about other
3682 applications using the token. If this is the case, then the consequences of the **C_InitToken** call are
3683 undefined.

3684 The **C_InitToken** function may not be sufficient to properly initialize complex tokens. In these situations,
3685 an initialization mechanism outside the scope of Cryptoki **MUST** be employed. The definition of "complex
3686 token" is product specific.

3687 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
3688 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
3689 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_PIN_INCORRECT**,
3690 **CKR_PIN_LOCKED**, **CKR_SESSION_EXISTS**, **CKR_SLOT_ID_INVALID**,
3691 **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**,
3692 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

3693 Example:

```
3694 CK_SLOT_ID slotID;  
3695 CK_UTF8CHAR pin[] = {"MyPIN"};  
3696 CK_UTF8CHAR label[32];  
3697 CK_RV rv;  
3698  
3699 .  
3700 .  
3701 memset(label, ' ', sizeof(label));  
3702 memcpy(label, "My first token", strlen("My first token"));  
3703 rv = C_InitToken(slotID, pin, strlen(pin), label);  
3704 if (rv == CKR_OK) {  
3705     .  
3706     .  
3707 }
```

3708 5.5.8 C_InitPIN

```
3709 CK_DECLARE_FUNCTION(CK_RV, C_InitPIN)(  
3710     CK_SESSION_HANDLE hSession,  
3711     CK_UTF8CHAR_PTR pPin,  
3712     CK_ULONG ulPinLen  
3713 );
```

3714 **C_InitPIN** initializes the normal user's PIN. *hSession* is the session's handle; *pPin* points to the normal
3715 user's PIN; *ulPinLen* is the length in bytes of the PIN. This standard allows PIN values to contain any
3716 valid UTF8 character, but the token may impose subset restrictions.

3717 **C_InitPIN** can only be called in the "R/W SO Functions" state. An attempt to call it from a session in any
3718 other state fails with error **CKR_USER_NOT_LOGGED_IN**.

3719 If the token has a "protected authentication path", as indicated by the
3720 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
3721 that there is some way for a user to be authenticated to the token without having to send a PIN through
3722 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3723 on the slot device. To initialize the normal user's PIN on a token with such a protected authentication
3724 path, the *pPin* parameter to **C_InitPIN** should be **NULL_PTR**. During the execution of **C_InitPIN**, the SO
3725 will enter the new PIN through the protected authentication path.

3726 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
3727 or not **C_InitPIN** can be used to initialize the normal user's token access.

3728 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
3729 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
3730 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_OPERATION_ACTIVE**,
3731 **CKR_PENDING**, **CKR_PIN_INVALID**, **CKR_PIN_LEN_RANGE**, **CKR_SESSION_CLOSED**,
3732 **CKR_SESSION_READ_ONLY**, **CKR_SESSION_HANDLE_INVALID**,
3733 **CKR_TOKEN_WRITE_PROTECTED**, **CKR_USER_NOT_LOGGED_IN**, **CKR_ARGUMENTS_BAD**.

3734 Example:

```
3735 CK_SESSION_HANDLE hSession;  
3736 CK_UTF8CHAR newPin[] = {"NewPIN"};  
3737 CK_RV rv;  
3738  
3739 rv = C_InitPIN(hSession, newPin, sizeof(newPin)-1);  
3740 if (rv == CKR_OK) {  
3741     .  
3742     .  
3743 }
```

3744 5.5.9 C_SetPIN

```
3745 CK_DECLARE_FUNCTION(CK_RV, C_SetPIN) (  
3746     CK_SESSION_HANDLE hSession,  
3747     CK_UTF8CHAR_PTR pOldPin,  
3748     CK_ULONG ulOldLen,  
3749     CK_UTF8CHAR_PTR pNewPin,  
3750     CK_ULONG ulNewLen  
3751 );
```

3752 **C_SetPIN** modifies the PIN of the user that is currently logged in, or the **CKU_USER** PIN if the session is
3753 not logged in. *hSession* is the session's handle; *pOldPin* points to the old PIN; *ulOldLen* is the length in
3754 bytes of the old PIN; *pNewPin* points to the new PIN; *ulNewLen* is the length in bytes of the new PIN. This
3755 standard allows PIN values to contain any valid UTF8 character, but the token may impose subset
3756 restrictions.

3757 **C_SetPIN** can only be called in the "R/W Public Session" state, "R/W SO Functions" state, or "R/W User
3758 Functions" state. An attempt to call it from a session in any other state fails with error
3759 **CKR_SESSION_READ_ONLY**.

3760 If the token has a "protected authentication path", as indicated by the
3761 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
3762 that there is some way for a user to be authenticated to the token without having to send a PIN through

3763 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
3764 on the slot device. To modify the current user's PIN on a token with such a protected authentication path,
3765 the *pOldPin* and *pNewPin* parameters to **C_SetPIN** should be **NULL_PTR**. During the execution of
3766 **C_SetPIN**, the current user will enter the old PIN and the new PIN through the protected authentication
3767 path. It is not specified how the PIN pad should be used to enter *two* PINs; this varies.

3768 If the token has a protected authentication path other than a PIN pad, then it is token-dependent whether
3769 or not **C_SetPIN** can be used to modify the current user's PIN.

3770 Return values: **CKR_CRYPTOKI_NOT_INITIALIZED**, **CKR_DEVICE_ERROR**, **CKR_DEVICE_MEMORY**,
3771 **CKR_DEVICE_REMOVED**, **CKR_FUNCTION_CANCELED**, **CKR_FUNCTION_FAILED**,
3772 **CKR_GENERAL_ERROR**, **CKR_HOST_MEMORY**, **CKR_OK**, **CKR_OPERATION_ACTIVE**,
3773 **CKR_PENDING**, **CKR_PIN_INCORRECT**, **CKR_PIN_INVALID**, **CKR_PIN_LEN_RANGE**,
3774 **CKR_PIN_LOCKED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**,
3775 **CKR_SESSION_READ_ONLY**, **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

3776 Example:

```
3777 CK_SESSION_HANDLE hSession;  
3778 CK_UTF8CHAR oldPin[] = {"OldPIN"};  
3779 CK_UTF8CHAR newPin[] = {"NewPIN"};  
3780 CK_RV rv;  
3781  
3782 rv = C_SetPIN(  
3783     hSession, oldPin, sizeof(oldPin)-1, newPin, sizeof(newPin)-1);  
3784 if (rv == CKR_OK) {  
3785     .  
3786     .  
3787 }
```

3788 5.6 Session management functions

3789 A typical application might perform the following series of steps to make use of a token (note that there
3790 are other reasonable sequences of events that an application might perform):

- 3791 1. Select a token.
- 3792 2. Make one or more calls to **C_OpenSession** to obtain one or more sessions with the token.
- 3793 3. Call **C_Login** to log the user into the token. Since all sessions an application has with a token have a
3794 shared login state, **C_Login** only needs to be called for one of the sessions.
- 3795 4. Perform cryptographic operations using the sessions with the token.
- 3796 5. Call **C_CloseSession** once for each session that the application has with the token, or call
3797 **C_CloseAllSessions** to close all the application's sessions simultaneously.

3798 As has been observed, an application may have concurrent sessions with more than one token. It is also
3799 possible for a token to have concurrent sessions with more than one application.

3800 Cryptoki provides the following functions for session management:

3801 5.6.1 C_OpenSession

```
3802 CK_DECLARE_FUNCTION(CK_RV, C_OpenSession) (  
3803     CK_SLOT_ID slotID,  
3804     CK_FLAGS flags,  
3805     CK_VOID_PTR pApplication,  
3806     CK_NOTIFY Notify,
```

```

3807     CK_SESSION_HANDLE_PTR phSession
3808 );

```

3809 **C_OpenSession** opens a session between an application and a token in a particular slot. *slotID* is the slot's ID; *flags* indicates the type of session; *pApplication* is an application-defined pointer to be passed to the notification callback; *Notify* is the address of the notification callback function (see Section 5.21); *phSession* points to the location that receives the handle for the new session.

3813 When opening a session with **C_OpenSession**, the *flags* parameter consists of the logical OR of zero or more bit flags defined in the **CK_SESSION_INFO** data type. For legacy reasons, the **CKF_SERIAL_SESSION** bit MUST always be set; if a call to **C_OpenSession** does not have this bit set, the call should return unsuccessfully with the error code **CKR_SESSION_PARALLEL_NOT_SUPPORTED**.

3818 There may be a limit on the number of concurrent sessions an application may have with the token, which may depend on whether the session is "read-only" or "read/write". An attempt to open a session which does not succeed because there are too many existing sessions of some type should return **CKR_SESSION_COUNT**.

3822 If the token is write-protected (as indicated in the **CK_TOKEN_INFO** structure), then only read-only sessions may be opened with it.

3824 If the application calling **C_OpenSession** already has a R/W SO session open with the token, then any attempt to open a R/O session with the token fails with error code **CKR_SESSION_READ_WRITE_SO_EXISTS** (see [PKCS11-UG] for further details).

3827 The *Notify* callback function is used by Cryptoki to notify the application of certain events. If the application does not wish to support callbacks, it should pass a value of **NULL_PTR** as the *Notify* parameter. See Section 5.21 for more information about application callbacks.

3830 As of version 3.2 an application can request an asynchronous session by providing the **CKF_ASYNC_SESSION** flag in the *flags* parameter. If the token does not support asynchronous operations, it should return **CKR_SESSION_ASYNC_NOT_SUPPORTED**. Tokens must support synchronous sessions. Tokens may support asynchronous sessions and may return **CKR_PENDING** if the token determines that the operation will take a long time to conclude.

3835 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_ASYNC_NOT_SUPPORTED**, **CKR_SESSION_COUNT**, **CKR_SESSION_PARALLEL_NOT_SUPPORTED**, **CKR_SESSION_READ_WRITE_SO_EXISTS**, **CKR_SLOT_ID_INVALID**, **CKR_TOKEN_NOT_PRESENT**, **CKR_TOKEN_NOT_RECOGNIZED**, **CKR_TOKEN_WRITE_PROTECTED**, **CKR_ARGUMENTS_BAD**.

3842 Example: see **C_CloseSession**.

3843 5.6.2 C_CloseSession

```

3844 CK_DECLARE_FUNCTION(CK_RV, C_CloseSession)(
3845     CK_SESSION_HANDLE hSession
3846 );

```

3847 **C_CloseSession** closes a session between an application and a token. *hSession* is the session's handle.

3848 **C_CloseSession** cancels all pending operations whose ID has not been retrieved using **C_AsyncGetID**. After a successful call to **C_CloseSession** the caller should free any memory passed into functions that returned **CKR_PENDING**.

3851 When a session is closed, all session objects created by the session are destroyed automatically, even if the application has other sessions "using" the objects (see [PKCS11-UG] for further details).

3853 If this function is successful and it closes the last session between the application and the token, the login state of the token for the application returns to public sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W Public sessions.

3856 Depending on the token, when the last open session any application has with the token is closed, the
3857 token may be “ejected” from its reader (if this capability exists).

3858 Despite the fact this **C_CloseSession** is supposed to close a session, the return value
3859 **CKR_SESSION_CLOSED** is an *error* return. It actually indicates the (probably somewhat unlikely) event
3860 that while this function call was executing, another call was made to **C_CloseSession** to close this
3861 particular session, and that call finished executing first. Such uses of sessions are a bad idea, and
3862 Cryptoki makes little promise of what will occur in general if an application indulges in this sort of
3863 behavior.

3864 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3865 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3866 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
3867 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

3868 Example:

```
3869 CK_SLOT_ID slotID;  
3870 CK_BYTE application;  
3871 CK_NOTIFY MyNotify;  
3872 CK_SESSION_HANDLE hSession;  
3873 CK_RV rv;  
3874  
3875 .  
3876 .  
3877 application = 17;  
3878 MyNotify = &EncryptionSessionCallback;  
3879 rv = C_OpenSession(  
3880     slotID, CKF_SERIAL_SESSION | CKF_RW_SESSION,  
3881     (CK_VOID_PTR) &application, MyNotify,  
3882     &hSession);  
3883 if (rv == CKR_OK) {  
3884     .  
3885     .  
3886     C_CloseSession(hSession);  
3887 }
```

3888 5.6.3 C_CloseAllSessions

```
3889 CK_DECLARE_FUNCTION(CK_RV, C_CloseAllSessions)(  
3890     CK_SLOT_ID slotID  
3891 );
```

3892 **C_CloseAllSessions** closes all sessions an application has with a token. *slotID* specifies the token’s slot.

3893 **C_CloseAllSessions** cancels all pending operations whose ID has not been retrieved using
3894 **C_AsyncGetID**. After a successful call to **C_CloseAllSessions** the caller should free any memory
3895 passed into functions that returned **CKR_PENDING**.

3896 When a session is closed, all session objects created by the session are destroyed automatically.

3897 After successful execution of this function, the login state of the token for the application returns to public
3898 sessions. Any new sessions to the token opened by the application will be either R/O Public or R/W
3899 Public sessions.

3900 Depending on the token, when the last open session any application has with the token is closed, the
3901 token may be “ejected” from its reader (if this capability exists).
3902 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3903 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3904 CKR_HOST_MEMORY, CKR_OK, CKR_SLOT_ID_INVALID, CKR_TOKEN_NOT_PRESENT.

3905 Example:

```
3906 CK_SLOT_ID slotID;  
3907 CK_RV rv;  
3908  
3909 .  
3910 .  
3911 rv = C_CloseAllSessions(slotID);
```

3912 5.6.4 C_GetSessionInfo

```
3913 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionInfo) (  
3914     CK_SESSION_HANDLE hSession,  
3915     CK_SESSION_INFO_PTR pInfo  
3916 );
```

3917 **C_GetSessionInfo** obtains information about a session. *hSession* is the session’s handle; *pInfo* points to
3918 the location that receives the session information.

3919 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3920 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3921 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
3922 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

3923 Example:

```
3924 CK_SESSION_HANDLE hSession;  
3925 CK_SESSION_INFO info;  
3926 CK_RV rv;  
3927  
3928 .  
3929 .  
3930 rv = C_GetSessionInfo(hSession, &info);  
3931 if (rv == CKR_OK) {  
3932     if (info.state == CKS_RW_USER_FUNCTIONS) {  
3933         .  
3934         .  
3935     }  
3936     .  
3937     .  
3938 }
```

3939 5.6.5 C_SessionCancel

```
3940 CK_DECLARE_FUNCTION(CK_RV, C_SessionCancel) (  
3941     CK_SESSION_HANDLE hSession
```

```
3942     CK_FLAGS flags
3943 );
```

3944 **C_SessionCancel** terminates active session based operations. *hSession* is the session's handle; *flags*
3945 indicates the operations to cancel.

3946 To identify which operation(s) should be terminated, the *flags* parameter should be assigned the logical
3947 bitwise OR of one or more of the bit flags defined in the **CK_MECHANISM_INFO** structure.

3948 **C_SessionCancel** will cancel a pending operation matching the input flags including those whose ID has
3949 been retrieved using **C_AsyncGetID**.

3950 If no flags are set, the session state will not be modified and CKR_OK will be returned.

3951 If a flag is set for an operation that has not been initialized in the session, the operation flag will be
3952 ignored and **C_SessionCancel** will behave as if the operation flag was not set.

3953 If any of the operations indicated by the *flags* parameter cannot be cancelled,
3954 CKR_OPERATION_CANCEL_FAILED must be returned. If multiple operation flags were set and
3955 CKR_OPERATION_CANCEL_FAILED is returned, this function does not provide any information about
3956 which operation(s) could not be cancelled. If an application desires to know if any single operation could
3957 not be cancelled, the application should not call **C_SessionCancel** with multiple flags set.

3958 If **C_SessionCancel** is called from an application callback (see Section 5.21), no action will be taken by
3959 the library and CKR_FUNCTION_FAILED must be returned.

3960 If **C_SessionCancel** is used to cancel one half of a dual-function operation, the remaining operation
3961 should still be left in an active state. However, it is expected that some Cryptoki implementations may not
3962 support this and return CKR_OPERATION_CANCEL_FAILED unless flags for both operations are
3963 provided.

3964 After a successful call to **C_SessionCancel** the caller should free any memory passed into functions that
3965 returned CKR_PENDING and were canceled.

3966 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
3967 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
3968 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_CANCEL_FAILED,
3969 CKR_PENDING, CKR_TOKEN_NOT_PRESENT.

3970 Example:

```
3971 CK_SESSION_HANDLE hSession;
3972 CK_RV rv;
3973
3974 rv = C_EncryptInit(hSession, &mechanism, hKey);
3975 if (rv != CKR_OK)
3976 {
3977     .
3978     .
3979 }
3980
3981 rv = C_SessionCancel (hSession, CKF_ENCRYPT);
3982 if (rv != CKR_OK)
3983 {
3984     .
3985     .
3986 }
3987
3988 rv = C_EncryptInit(hSession, &mechanism, hKey);
```

```

3989 if (rv != CKR_OK)
3990 {
3991     .
3992     .
3993 }
3994

```

3995
3996 Below are modifications to existing API descriptions to allow an alternate method of cancelling individual
3997 operations. The additional text is highlighted.

3998 5.6.6 C_GetOperationState

```

3999 CK_DECLARE_FUNCTION(CK_RV, C_GetOperationState) (
4000     CK_SESSION_HANDLE hSession,
4001     CK_BYTE_PTR pOperationState,
4002     CK_ULONG_PTR pulOperationStateLen
4003 );

```

4004 **C_GetOperationState** obtains a copy of the cryptographic operations state of a session, encoded as a
4005 string of bytes. *hSession* is the session's handle; *pOperationState* points to the location that receives the
4006 state; *pulOperationStateLen* points to the location that receives the length in bytes of the state.

4007 Although the saved state output by **C_GetOperationState** is not really produced by a “cryptographic
4008 mechanism”, **C_GetOperationState** nonetheless uses the convention described in Section 5.2 on
4009 producing output.

4010 Precisely what the “cryptographic operations state” this function saves is varies from token to token;
4011 however, this state is what is provided as input to **C_SetOperationState** to restore the cryptographic
4012 activities of a session.

4013 Consider a session which is performing a message digest operation using SHA-1 (*i.e.*, the session is
4014 using the **CKM_SHA_1** mechanism). Suppose that the message digest operation was initialized properly,
4015 and that precisely 80 bytes of data have been supplied so far as input to SHA-1. The application now
4016 wants to “save the state” of this digest operation, so that it can continue it later. In this particular case,
4017 since SHA-1 processes 512 bits (64 bytes) of input at a time, the cryptographic operations state of the
4018 session most likely consists of three distinct parts: the state of SHA-1's 160-bit internal chaining variable;
4019 the 16 bytes of unprocessed input data; and some administrative data indicating that this saved state
4020 comes from a session which was performing SHA-1 hashing. Taken together, these three pieces of
4021 information suffice to continue the current hashing operation at a later time.

4022 Consider next a session which is performing an encryption operation with DES (a block cipher with a
4023 block size of 64 bits) in CBC (cipher-block chaining) mode (*i.e.*, the session is using the **CKM_DES_CBC**
4024 mechanism). Suppose that precisely 22 bytes of data (in addition to an IV for the CBC mode) have been
4025 supplied so far as input to DES, which means that the first two 8-byte blocks of ciphertext have already
4026 been produced and output. In this case, the cryptographic operations state of the session most likely
4027 consists of three or four distinct parts: the second 8-byte block of ciphertext (this will be used for cipher-
4028 block chaining to produce the next block of ciphertext); the 6 bytes of data still awaiting encryption; some
4029 administrative data indicating that this saved state comes from a session which was performing DES
4030 encryption in CBC mode; and possibly the DES key being used for encryption (see **C_SetOperationState**
4031 for more information on whether or not the key is present in the saved state).

4032 If a session is performing two cryptographic operations simultaneously (see Section 5.14), then the
4033 cryptographic operations state of the session will contain all the necessary information to restore both
4034 operations.

4035 An attempt to save the cryptographic operations state of a session which does not currently have some
4036 active savable cryptographic operation(s) (encryption, decryption, digesting, signing without message
4037 recovery, verification without message recovery, or some legal combination of two of these) should fail
4038 with the error CKR_OPERATION_NOT_INITIALIZED.

4039 An attempt to save the cryptographic operations state of a session which is performing an appropriate
4040 cryptographic operation (or two), but which cannot be satisfied for any of various reasons (certain
4041 necessary state information and/or key information can't leave the token, for example) should fail with the
4042 error CKR_STATE_UNSAVEABLE.

4043 Return values: CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
4044 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4045 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4046 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
4047 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_STATE_UNSAVEABLE,
4048 CKR_ARGUMENTS_BAD.

4049 Example: see **C_SetOperationState**.

4050 5.6.7 C_SetOperationState

```
4051 CK_DECLARE_FUNCTION(CK_RV, C_SetOperationState) (  
4052     CK_SESSION_HANDLE hSession,  
4053     CK_BYTE_PTR pOperationState,  
4054     CK_ULONG ulOperationStateLen,  
4055     CK_OBJECT_HANDLE hEncryptionKey,  
4056     CK_OBJECT_HANDLE hAuthenticationKey  
4057 );
```

4058 **C_SetOperationState** restores the cryptographic operations state of a session from a string of bytes
4059 obtained with **C_GetOperationState**. *hSession* is the session's handle; *pOperationState* points to the
4060 location holding the saved state; *ulOperationStateLen* holds the length of the saved state;
4061 *hEncryptionKey* holds a handle to the key which will be used for an ongoing encryption or decryption
4062 operation in the restored session (or 0 if no encryption or decryption key is needed, either because no
4063 such operation is ongoing in the stored session or because all the necessary key information is present in
4064 the saved state); *hAuthenticationKey* holds a handle to the key which will be used for an ongoing
4065 signature, MACing, or verification operation in the restored session (or 0 if no such key is needed, either
4066 because no such operation is ongoing in the stored session or because all the necessary key information
4067 is present in the saved state).

4068 The state need not have been obtained from the same session (the "source session") as it is being
4069 restored to (the "destination session"). However, the source session and destination session should have
4070 a common session state (e.g., **CKS_RW_USER_FUNCTIONS**), and should be with a common token.
4071 There is also no guarantee that cryptographic operations state may be carried across logins, or across
4072 different Cryptoki implementations.

4073 If **C_SetOperationState** is supplied with alleged saved cryptographic operations state which it can
4074 determine is not valid saved state (or is cryptographic operations state from a session with a different
4075 session state, or is cryptographic operations state from a different token), it fails with the error
4076 CKR_SAVED_STATE_INVALID.

4077 Saved state obtained from calls to **C_GetOperationState** may or may not contain information about keys
4078 in use for ongoing cryptographic operations. If a saved cryptographic operations state has an ongoing
4079 encryption or decryption operation, and the key in use for the operation is not saved in the state, then it
4080 MUST be supplied to **C_SetOperationState** in the *hEncryptionKey* argument. If it is not, then
4081 **C_SetOperationState** will fail and return the error CKR_KEY_NEEDED. If the key in use for the
4082 operation is saved in the state, then it *can* be supplied in the *hEncryptionKey* argument, but this is not
4083 required.

4084 Similarly, if a saved cryptographic operations state has an ongoing signature, MACing, or verification
4085 operation, and the key in use for the operation is not saved in the state, then it MUST be supplied to
4086 **C_SetOperationState** in the *hAuthenticationKey* argument. If it is not, then **C_SetOperationState** will fail
4087 with the error CKR_KEY_NEEDED. If the key in use for the operation is saved in the state, then it *can* be
4088 supplied in the *hAuthenticationKey* argument, but this is not required.

4089 If an *irrelevant* key is supplied to **C_SetOperationState** call (e.g., a nonzero key handle is submitted in
4090 the *hEncryptionKey* argument, but the saved cryptographic operations state supplied does not have an

4091 ongoing encryption or decryption operation, then **C_SetOperationState** fails with the error
4092 CKR_KEY_NOT_NEEDED.

4093 If a key is supplied as an argument to **C_SetOperationState**, and **C_SetOperationState** can somehow
4094 detect that this key was not the key being used in the source session for the supplied cryptographic
4095 operations state (it may be able to detect this if the key or a hash of the key is present in the saved state,
4096 for example), then **C_SetOperationState** fails with the error CKR_KEY_CHANGED.

4097 An application can look at the **CKF_RESTORE_KEY_NOT_NEEDED** flag in the flags field of the
4098 **CK_TOKEN_INFO** field for a token to determine whether or not it needs to supply key handles to
4099 **C_SetOperationState** calls. If this flag is true, then a call to **C_SetOperationState** *never* needs a key
4100 handle to be supplied to it. If this flag is false, then at least some of the time, **C_SetOperationState**
4101 requires a key handle, and so the application should probably *always* pass in any relevant key handles
4102 when restoring cryptographic operations state to a session.

4103 **C_SetOperationState** can successfully restore cryptographic operations state to a session even if that
4104 session has active cryptographic or object search operations when **C_SetOperationState** is called (the
4105 ongoing operations are abruptly cancelled).

4106 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4107 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4108 CKR_HOST_MEMORY, CKR_KEY_CHANGED, CKR_KEY_NEEDED, CKR_KEY_NOT_NEEDED,
4109 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_SAVED_STATE_INVALID,
4110 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD.

4111 Example:

```

4112 CK_SESSION_HANDLE hSession;
4113 CK_MECHANISM digestMechanism;
4114 CK_BYTE_PTR pState;
4115 CK_ULONG ulStateLen;
4116 CK_BYTE data1[] = {0x01, 0x03, 0x05, 0x07};
4117 CK_BYTE data2[] = {0x02, 0x04, 0x08};
4118 CK_BYTE data3[] = {0x10, 0x0F, 0x0E, 0x0D, 0x0C};
4119 CK_BYTE pDigest[20];
4120 CK_ULONG ulDigestLen;
4121 CK_RV rv;
4122
4123 .
4124 .
4125 /* Initialize hash operation */
4126 rv = C_DigestInit(hSession, &digestMechanism);
4127 assert(rv == CKR_OK);
4128
4129 /* Start hashing */
4130 rv = C_DigestUpdate(hSession, data1, sizeof(data1));
4131 assert(rv == CKR_OK);
4132
4133 /* Find out how big the state might be */
4134 rv = C_GetOperationState(hSession, NULL_PTR, &ulStateLen);
4135 assert(rv == CKR_OK);
4136

```

```

4137 /* Allocate some memory and then get the state */
4138 pState = (CK_BYTE_PTR) malloc(ulStateLen);
4139 rv = C_GetOperationState(hSession, pState, &ulStateLen);
4140
4141 /* Continue hashing */
4142 rv = C_DigestUpdate(hSession, data2, sizeof(data2));
4143 assert(rv == CKR_OK);
4144
4145 /* Restore state. No key handles needed */
4146 rv = C_SetOperationState(hSession, pState, ulStateLen, 0, 0);
4147 assert(rv == CKR_OK);
4148
4149 /* Continue hashing from where we saved state */
4150 rv = C_DigestUpdate(hSession, data3, sizeof(data3));
4151 assert(rv == CKR_OK);
4152
4153 /* Conclude hashing operation */
4154 ulDigestLen = sizeof(pDigest);
4155 rv = C_DigestFinal(hSession, pDigest, &ulDigestLen);
4156 if (rv == CKR_OK) {
4157     /* pDigest[] now contains the hash of 0x01030507100F0E0D0C */
4158     .
4159     .
4160 }

```

4161 5.6.8 C_Login

```

4162 CK_DECLARE_FUNCTION(CK_RV, C_Login)(
4163     CK_SESSION_HANDLE hSession,
4164     CK_USER_TYPE userType,
4165     CK_UTF8CHAR_PTR pPin,
4166     CK_ULONG ulPinLen
4167 );

```

4168 **C_Login** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin* points to
4169 the user's PIN; *ulPinLen* is the length of the PIN. This standard allows PIN values to contain any valid
4170 UTF8 character, but the token may impose subset restrictions.

4171 When the user type is either **CKU_SO** or **CKU_USER**, if the call succeeds, each of the application's
4172 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
4173 Functions" state. If the user type is **CKU_CONTEXT_SPECIFIC**, the behavior of **C_Login** depends on
4174 the context in which it is called. Improper use of this user type will result in a return value
4175 **CKR_OPERATION_NOT_INITIALIZED**.

4176 If the token has a "protected authentication path", as indicated by the
4177 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its **CK_TOKEN_INFO** being set, then that means
4178 that there is some way for a user to be authenticated to the token without having to send a PIN through
4179 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
4180 on the slot device. Or the user might not even use a PIN—authentication could be achieved by some
4181 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
4182 parameter to **C_Login** should be **NULL_PTR**. When **C_Login** returns, whatever authentication method

4183 supported by the token will have been performed; a return value of CKR_OK means that the user was
4184 successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
4185 denied access.

4186 If there are any active cryptographic or object finding operations in an application's session, and then
4187 **C_Login** is successfully executed by that application, it may or may not be the case that those operations
4188 are still active. Therefore, before logging in, any active operations should be finished.

4189 If the application calling **C_Login** has a R/O session open with the token, then it will be unable to log the
4190 SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error code
4191 CKR_SESSION_READ_ONLY_EXISTS.

4192 C_Login may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
4193 **CKA_ALWAYS_AUTHENTICATE** attribute set to CK_TRUE exists, and the user needs to do
4194 cryptographic operation on this key. See further Section 4.10.

4195 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4196 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4197 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4198 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
4199 CKR_PENDING, CKR_PIN_INCORRECT, CKR_PIN_LOCKED, CKR_SESSION_CLOSED,
4200 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY_EXISTS,
4201 CKR_USER_ALREADY_LOGGED_IN, CKR_USER_ANOTHER_ALREADY_LOGGED_IN,
4202 CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

4203 Example: see **C_Logout**.

4204 5.6.9 C_LoginUser

```
4205 CK_DECLARE_FUNCTION(CK_RV, C_LoginUser) (  
4206     CK_SESSION_HANDLE hSession,  
4207     CK_USER_TYPE userType,  
4208     CK_UTF8CHAR_PTR pPin,  
4209     CK_ULONG ulPinLen,  
4210     CK_UTF8CHAR_PTR pUsername,  
4211     CK_ULONG ulUsernameLen  
4212 );
```

4213 **C_LoginUser** logs a user into a token. *hSession* is a session handle; *userType* is the user type; *pPin*
4214 points to the user's PIN; *ulPinLen* is the length of the PIN, *pUsername* points to the user name,
4215 *ulUsernameLen* is the length of the user name. This standard allows PIN and user name values to
4216 contain any valid UTF8 character, but the token may impose subset restrictions.

4217 When the user type is either **CKU_SO** or **CKU_USER**, if the call succeeds, each of the application's
4218 sessions will enter either the "R/W SO Functions" state, the "R/W User Functions" state, or the "R/O User
4219 Functions" state. If the user type is **CKU_CONTEXT_SPECIFIC**, the behavior of **C_LoginUser** depends
4220 on the context in which it is called. Improper use of this user type will result in a return value
4221 CKR_OPERATION_NOT_INITIALIZED.

4222 If the token has a "protected authentication path", as indicated by the
4223 **CKF_PROTECTED_AUTHENTICATION_PATH** flag in its CK_TOKEN_INFO being set, then that means
4224 that there is some way for a user to be authenticated to the token without having to send a PIN through
4225 the Cryptoki library. One such possibility is that the user enters a PIN on a PIN pad on the token itself, or
4226 on the slot device. The user might not even use a PIN—authentication could be achieved by some
4227 fingerprint-reading device, for example. To log into a token with a protected authentication path, the *pPin*
4228 parameter to **C_LoginUser** should be NULL_PTR. When **C_LoginUser** returns, whatever authentication
4229 method supported by the token will have been performed; a return value of CKR_OK means that the user
4230 was successfully authenticated, and a return value of CKR_PIN_INCORRECT means that the user was
4231 denied access.

4232 If there are any active cryptographic or object finding operations in an application's session, and then
4233 **C_LoginUser** is successfully executed by that application, it may or may not be the case that those
4234 operations are still active. Therefore, before logging in, any active operations should be finished.

4235 If the application calling **C_LoginUser** has a R/O session open with the token, then it will be unable to log
4236 the SO into a session (see [PKCS11-UG] for further details). An attempt to do this will result in the error
4237 code CKR_SESSION_READ_ONLY_EXISTS.

4238 **C_LoginUser** may be called repeatedly, without intervening **C_Logout** calls, if (and only if) a key with the
4239 **CKA_ALWAYS_AUTHENTICATE** attribute set to CK_TRUE exists, and the user needs to do
4240 cryptographic operation on this key. See further Section 4.10.

4241 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4242 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4243 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4244 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
4245 CKR_PENDING, CKR_PIN_INCORRECT, CKR_PIN_LOCKED, CKR_SESSION_CLOSED,
4246 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY_EXISTS,
4247 CKR_USER_ALREADY_LOGGED_IN, CKR_USER_ANOTHER_ALREADY_LOGGED_IN,
4248 CKR_USER_PIN_NOT_INITIALIZED, CKR_USER_TOO_MANY_TYPES, CKR_USER_TYPE_INVALID.

4249 Example:

```
4250 CK_SESSION_HANDLE hSession;  
4251 CK_UTF8CHAR userPin[] = {"MyPIN"};  
4252 CK_UTF8CHAR userName[] = {"MyUserName"};  
4253 CK_RV rv;  
4254  
4255 rv = C_LoginUser(hSession, CKU_USER, userPin, sizeof(userPin)-1, userName,  
4256 sizeof(userName)-1);  
4257 if (rv == CKR_OK) {  
4258     .  
4259     .  
4260     rv = C_Logout(hSession);  
4261     if (rv == CKR_OK) {  
4262         .  
4263         .  
4264     }  
4265 }
```

4266 5.6.10 C_Logout

```
4267 CK_DECLARE_FUNCTION(CK_RV, C_Logout) (  
4268     CK_SESSION_HANDLE hSession  
4269 );
```

4270 **C_Logout** logs a user out from a token. *hSession* is the session's handle.

4271 Depending on the current user type, if the call succeeds, each of the application's sessions will enter
4272 either the "R/W Public Session" state or the "R/O Public Session" state.

4273 When **C_Logout** successfully executes, any of the application's handles to private objects become invalid
4274 (even if a user is later logged back into the token, those handles remain invalid). In addition, all private
4275 session objects from sessions belonging to the application are destroyed.

4276 If there are any active cryptographic or object-finding operations in an application's session, and then
4277 **C_Logout** is successfully executed by that application, it may or may not be the case that those
4278 operations are still active. Therefore, before logging out, any active operations should be finished.

4279 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4280 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,

4281 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
4282 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

4283 Example:

```
4284 CK_SESSION_HANDLE hSession;  
4285 CK_UTF8CHAR userPin[] = {"MyPIN"};  
4286 CK_RV rv;  
4287  
4288 rv = C_Login(hSession, CKU_USER, userPin, sizeof(userPin)-1);  
4289 if (rv == CKR_OK) {  
4290     .  
4291     .  
4292     rv = C_Logout(hSession);  
4293     if (rv == CKR_OK) {  
4294         .  
4295         .  
4296     }  
4297 }
```

4298 5.6.11 C_GetSessionValidationFlags

```
4299 CK_DECLARE_FUNCTION(CK_RV, C_GetSessionValidationFlags) (  
4300     CK_SESSION_HANDLE hSession,  
4301     CK_SESSION_VALIDATION_FLAGS_TYPE type,  
4302     CK_FLAGS_PTR pFlags,  
4303 );
```

4304 **C_GetSessionValidationFlags** fetches the requested flags from the session. See Validation indicators
4305 (section 4.15.3.1) for meaning and semantics for these flags. Applications are responsible for the
4306 appropriate locking to protect session to get a meaningful result from this call.

4307 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
4308 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4309 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
4310 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4311 5.7 Object management functions

4312 Cryptoki provides the following functions for managing objects. Additional functions provided specifically
4313 for managing key objects are described in Section 5.18.

4314 5.7.1 C_CreateObject

```
4315 CK_DECLARE_FUNCTION(CK_RV, C_CreateObject) (  
4316     CK_SESSION_HANDLE hSession,  
4317     CK_ATTRIBUTE_PTR pTemplate,  
4318     CK_ULONG ulCount,  
4319     CK_OBJECT_HANDLE_PTR phObject  
4320 );
```

4321 **C_CreateObject** creates a new object. *hSession* is the session's handle; *pTemplate* points to the object's
4322 template; *ulCount* is the number of attributes in the template; *phObject* points to the location that receives
4323 the new object's handle.

4324 If a call to **C_CreateObject** cannot support the precise template supplied to it, it will fail and return without
4325 creating any object.

4326 If **C_CreateObject** is used to create a key object, the key object will have its **CKA_LOCAL** attribute set to
4327 CK_FALSE. If that key object is a secret or private key then the new key will have the
4328 **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the **CKA_NEVER_EXTRACTABLE**
4329 attribute set to CK_FALSE.

4330 Only session objects can be created during a read-only session. Only public objects can be created
4331 unless the normal user is logged in.

4332 Whenever an object is created, a value for **CKA_UNIQUE_ID** is generated and assigned to the new
4333 object (See Section 4.4.1).

4334 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
4335 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
4336 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
4337 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
4338 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4339 CKR_OPERATION_ACTIVE, CKR_PARAMETER_SET_NOT_SUPPORTED, CKR_PENDING,
4340 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4341 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
4342 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

4343 Example:

```
4344 CK_SESSION_HANDLE hSession;  
4345 CK_OBJECT_HANDLE  
4346     hData,  
4347     hCertificate,  
4348     hKey;  
4349 CK_OBJECT_CLASS  
4350     dataClass = CKO_DATA,  
4351     certificateClass = CKO_CERTIFICATE,  
4352     keyClass = CKO_PUBLIC_KEY;  
4353 CK_KEY_TYPE keyType = CKK_RSA;  
4354 CK_UTF8CHAR application[] = {"My Application"};  
4355 CK_BYTE dataValue[] = {...};  
4356 CK_BYTE subject[] = {...};  
4357 CK_BYTE id[] = {...};  
4358 CK_BYTE certificateValue[] = {...};  
4359 CK_BYTE modulus[] = {...};  
4360 CK_BYTE exponent[] = {...};  
4361 CK_BBOOL true = CK_TRUE;  
4362 CK_ATTRIBUTE dataTemplate[] = {  
4363     {CKA_CLASS, &dataClass, sizeof(dataClass)},  
4364     {CKA_TOKEN, &true, sizeof(true)},  
4365     {CKA_APPLICATION, application, sizeof(application)-1},  
4366     {CKA_VALUE, dataValue, sizeof(dataValue)}  
4367 };  
4368 CK_ATTRIBUTE certificateTemplate[] = {  
4369     {CKA_CLASS, &certificateClass, sizeof(certificateClass)},
```

```

4370     {CKA_TOKEN, &true, sizeof(true)},
4371     {CKA_SUBJECT, subject, sizeof(subject)},
4372     {CKA_ID, id, sizeof(id)},
4373     {CKA_VALUE, certificateValue, sizeof(certificateValue)}
4374 };
4375 CK_ATTRIBUTE keyTemplate[] = {
4376     {CKA_CLASS, &keyClass, sizeof(keyClass)},
4377     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
4378     {CKA_WRAP, &true, sizeof(true)},
4379     {CKA_MODULUS, modulus, sizeof(modulus)},
4380     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
4381 };
4382 CK_RV rv;
4383
4384 .
4385 .
4386 /* Create a data object */
4387 rv = C_CreateObject(hSession, dataTemplate, 4, &hData);
4388 if (rv == CKR_OK) {
4389     .
4390     .
4391 }
4392
4393 /* Create a certificate object */
4394 rv = C_CreateObject(
4395     hSession, certificateTemplate, 5, &hCertificate);
4396 if (rv == CKR_OK) {
4397     .
4398     .
4399 }
4400
4401 /* Create an RSA public key object */
4402 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
4403 if (rv == CKR_OK) {
4404     .
4405     .
4406 }

```

4407 5.7.2 C_CopyObject

```

4408 CK_DECLARE_FUNCTION(CK_RV, C_CopyObject)(
4409     CK_SESSION_HANDLE hSession,
4410     CK_OBJECT_HANDLE hObject,
4411     CK_ATTRIBUTE_PTR pTemplate,
4412     CK_ULONG ulCount,

```

```
4413     CK_OBJECT_HANDLE_PTR phNewObject
4414 );
```

4415 **C_CopyObject** copies an object, creating a new object for the copy. *hSession* is the session's handle;
4416 *hObject* is the object's handle; *pTemplate* points to the template for the new object; *ulCount* is the number
4417 of attributes in the template; *phNewObject* points to the location that receives the handle for the copy of
4418 the object.

4419 The template may specify new values for any attributes of the object that can ordinarily be modified (e.g.,
4420 in the course of copying a secret key, a key's **CKA_EXTRACTABLE** attribute may be changed from
4421 CK_TRUE to CK_FALSE, but not the other way around. If this change is made, the new key's
4422 **CKA_NEVER_EXTRACTABLE** attribute will have the value CK_FALSE. Similarly, the template may
4423 specify that the new key's **CKA_SENSITIVE** attribute be CK_TRUE; the new key will have the same
4424 value for its **CKA_ALWAYS_SENSITIVE** attribute as the original key). It may also specify new values of
4425 the **CKA_TOKEN** and **CKA_PRIVATE** attributes (e.g., to copy a session object to a token object). If the
4426 template specifies a value of an attribute which is incompatible with other existing attributes of the object,
4427 the call fails with the return code CKR_TEMPLATE_INCONSISTENT.

4428 If a call to **C_CopyObject** cannot support the precise template supplied to it, it will fail and return without
4429 creating any object. If the object indicated by *hObject* has its **CKA_COPYABLE** attribute set to
4430 CK_FALSE, **C_CopyObject** will return CKR_ACTION_PROHIBITED.

4431 Whenever an object is copied, a new value for **CKA_UNIQUE_ID** is generated and assigned to the new
4432 object (See Section 4.4.1).

4433 Only session objects can be created during a read-only session. Only public objects can be created
4434 unless the normal user is logged in.

4435 Return values: , CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD,
4436 CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,
4437 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
4438 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
4439 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
4440 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
4441 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
4442 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
4443 CKR_USER_NOT_LOGGED_IN.

4444 Example:

```
4445 CK_SESSION_HANDLE hSession;
4446 CK_OBJECT_HANDLE hKey, hNewKey;
4447 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
4448 CK_KEY_TYPE keyType = CKK_DES;
4449 CK_BYTE id[] = {...};
4450 CK_BYTE keyValue[] = {...};
4451 CK_BBOOL false = CK_FALSE;
4452 CK_BBOOL true = CK_TRUE;
4453 CK_ATTRIBUTE keyTemplate[] = {
4454     {CKA_CLASS, &keyClass, sizeof(keyClass)},
4455     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
4456     {CKA_TOKEN, &>false, sizeof(false)},
4457     {CKA_ID, id, sizeof(id)},
4458     {CKA_VALUE, keyValue, sizeof(keyValue)}
4459 };
4460 CK_ATTRIBUTE copyTemplate[] = {
4461     {CKA_TOKEN, &>true, sizeof(true)}
```

```

4462 };
4463 CK_RV rv;
4464
4465 .
4466 .
4467 /* Create a DES secret key session object */
4468 rv = C_CreateObject(hSession, keyTemplate, 5, &hKey);
4469 if (rv == CKR_OK) {
4470     /* Create a copy which is a token object */
4471     rv = C_CopyObject(hSession, hKey, copyTemplate, 1, &hNewKey);
4472     .
4473     .
4474 }

```

4475 5.7.3 C_DestroyObject

```

4476 CK_DECLARE_FUNCTION(CK_RV, C_DestroyObject) (
4477     CK_SESSION_HANDLE hSession,
4478     CK_OBJECT_HANDLE hObject
4479 );

```

4480 **C_DestroyObject** destroys an object. *hSession* is the session's handle; and *hObject* is the object's handle.

4482 Only session objects can be destroyed during a read-only session. Only public objects can be destroyed unless the normal user is logged in.

4484 Certain objects may not be destroyed. Calling **C_DestroyObject** on such objects will result in the CKR_ACTION_PROHIBITED error code. An application can consult the object's **CKA_DESTROYABLE** attribute to determine if an object may be destroyed or not.

4487 Return values: CKR_ACTION_PROHIBITED, CKR_CRYPTOKI_NOT_INITIALIZED,
4488 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4489 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
4490 CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
4491 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
4492 CKR_SESSION_READ_ONLY, CKR_TOKEN_WRITE_PROTECTED.

4493 Example: see **C_GetObjectSize**.

4494 5.7.4 C_GetObjectSize

```

4495 CK_DECLARE_FUNCTION(CK_RV, C_GetObjectSize) (
4496     CK_SESSION_HANDLE hSession,
4497     CK_OBJECT_HANDLE hObject,
4498     CK_ULONG_PTR pulSize
4499 );

```

4500 **C_GetObjectSize** gets the size of an object in bytes. *hSession* is the session's handle; *hObject* is the object's handle; *pulSize* points to the location that receives the size in bytes of the object.

4502 Cryptoki does not specify what the precise meaning of an object's size is. Intuitively, it is some measure of how much token memory the object takes up. If an application deletes (say) a private object of size *S*, it might be reasonable to assume that the *ulFreePrivateMemory* field of the token's **CK_TOKEN_INFO** structure increases by approximately *S*.

4506 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
4507 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,

4508 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
4509 CKR_INFORMATION_SENSITIVE, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
4510 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_SESSION_CLOSED,
4511 CKR_SESSION_HANDLE_INVALID.

4512 Example:

```
4513 CK_SESSION_HANDLE hSession;  
4514 CK_OBJECT_HANDLE hObject;  
4515 CK_OBJECT_CLASS dataClass = CKO_DATA;  
4516 CK_UTF8CHAR application[] = {"My Application"};  
4517 CK_BYTE value[] = {...};  
4518 CK_BBOOL true = CK_TRUE;  
4519 CK_ATTRIBUTE template[] = {  
4520     {CKA_CLASS, &dataClass, sizeof(dataClass)},  
4521     {CKA_TOKEN, &true, sizeof(true)},  
4522     {CKA_APPLICATION, application, sizeof(application)-1},  
4523     {CKA_VALUE, value, sizeof(value)}  
4524 };  
4525 CK_ULONG ulSize;  
4526 CK_RV rv;  
4527  
4528 .  
4529 .  
4530 rv = C_CreateObject(hSession, template, 4, &hObject);  
4531 if (rv == CKR_OK) {  
4532     rv = C_GetObjectSize(hSession, hObject, &ulSize);  
4533     if (rv != CKR_INFORMATION_SENSITIVE) {  
4534         .  
4535         .  
4536     }  
4537  
4538     rv = C_DestroyObject(hSession, hObject);  
4539     .  
4540     .  
4541 }
```

4542 5.7.5 C_GetAttributeValue

```
4543 CK_DECLARE_FUNCTION(CK_RV, C_GetAttributeValue) (  
4544     CK_SESSION_HANDLE hSession,  
4545     CK_OBJECT_HANDLE hObject,  
4546     CK_ATTRIBUTE_PTR pTemplate,  
4547     CK_ULONG ulCount  
4548 );
```

4549 **C_GetAttributeValue** obtains the value of one or more attributes of an object. *hSession* is the session's
4550 handle; *hObject* is the object's handle; *pTemplate* points to a template that specifies which attribute

values are to be obtained, and receives the attribute values; *ulCount* is the number of attributes in the template.

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

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

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

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

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_SENSITIVE, CKR_ATTRIBUTE_TYPE_INVALID, CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```

4599 CK_SESSION_HANDLE hSession;
4600 CK_OBJECT_HANDLE hObject;
4601 CK_BYTE_PTR pModulus, pExponent;
4602 CK_ATTRIBUTE template[] = {

```

```

4603     {CKA_MODULUS, NULL_PTR, 0},
4604     {CKA_PUBLIC_EXPONENT, NULL_PTR, 0}
4605 };
4606 CK_RV rv;
4607
4608 .
4609 .
4610 rv = C_GetAttributeValue(hSession, hObject, template, 2);
4611 if (rv == CKR_OK) {
4612     pModulus = (CK_BYTE_PTR) malloc(template[0].ulValueLen);
4613     template[0].pValue = pModulus;
4614     /* template[0].ulValueLen was set by C_GetAttributeValue */
4615
4616     pExponent = (CK_BYTE_PTR) malloc(template[1].ulValueLen);
4617     template[1].pValue = pExponent;
4618     /* template[1].ulValueLen was set by C_GetAttributeValue */
4619
4620     rv = C_GetAttributeValue(hSession, hObject, template, 2);
4621     if (rv == CKR_OK) {
4622         .
4623         .
4624     }
4625     free(pModulus);
4626     free(pExponent);
4627 }

```

5.7.6 C_SetAttributeValue

```

4629 CK_DECLARE_FUNCTION(CK_RV, C_SetAttributeValue) (
4630     CK_SESSION_HANDLE hSession,
4631     CK_OBJECT_HANDLE hObject,
4632     CK_ATTRIBUTE_PTR pTemplate,
4633     CK_ULONG ulCount
4634 );

```

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

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

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

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

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

Return values: CKR_ACTION_PROHIBITED, CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID,

4648 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
 4649 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
 4650 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OBJECT_HANDLE_INVALID, CKR_OK,
 4651 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_SESSION_CLOSED,
 4652 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
 4653 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
 4654 CKR_USER_NOT_LOGGED_IN.

4655 Example:

```
4656 CK_SESSION_HANDLE hSession;
4657 CK_OBJECT_HANDLE hObject;
4658 CK_UTF8CHAR label[] = {"New label"};
4659 CK_ATTRIBUTE template[] = {
4660     {CKA_LABEL, label, sizeof(label)-1}
4661 };
4662 CK_RV rv;
4663
4664 .
4665 .
4666 rv = C_SetAttributeValue(hSession, hObject, template, 1);
4667 if (rv == CKR_OK) {
4668     .
4669     .
4670 }
```

4671 5.7.7 C_FindObjectsInit

```
4672 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsInit)(
4673     CK_SESSION_HANDLE hSession,
4674     CK_ATTRIBUTE_PTR pTemplate,
4675     CK_ULONG ulCount
4676 );
```

4677 **C_FindObjectsInit** initializes a search for token and session objects that match a template. *hSession* is
 4678 the session's handle; *pTemplate* points to a search template that specifies the attribute values to match;
 4679 *ulCount* is the number of attributes in the search template. The matching criterion is an exact byte-for-
 4680 byte match with all attributes in the template. To find all objects, set *ulCount* to 0.

4681 After calling **C_FindObjectsInit**, the application may call **C_FindObjects** one or more times to obtain
 4682 handles for objects matching the template, and then eventually call **C_FindObjectsFinal** to finish the
 4683 active search operation. At most one search operation may be active at a given time in a given session.

4684 The object search operation will only find objects that the session can view. For example, an object
 4685 search in an "R/W Public Session" will not find any private objects (even if one of the attributes in the
 4686 search template specifies that the search is for private objects).

4687 If a search operation is active, and objects are created or destroyed which fit the search template for the
 4688 active search operation, then those objects may or may not be found by the search operation. Note that
 4689 this means that, under these circumstances, the search operation may return invalid object handles.

4690 Even though **C_FindObjectsInit** can return the values CKR_ATTRIBUTE_TYPE_INVALID and
 4691 CKR_ATTRIBUTE_VALUE_INVALID, it is not required to. For example, if it is given a search template
 4692 with nonexistent attributes in it, it can return CKR_ATTRIBUTE_TYPE_INVALID, or it can initialize a
 4693 search operation which will match no objects and return CKR_OK.

4694 If the **CKA_UNIQUE_ID** attribute is present in the search template, either zero or one objects will be
 4695 found, since at most one object can have any particular **CKA_UNIQUE_ID** value.

4696 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_TYPE_INVALID,
 4697 CKR_ATTRIBUTE_VALUE_INVALID, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR,
 4698 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED,
 4699 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
 4700 CKR_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4701 Example: see **C_FindObjectsFinal**.

4702 5.7.8 C_FindObjects

```
4703 CK_DECLARE_FUNCTION(CK_RV, C_FindObjects)(
4704     CK_SESSION_HANDLE hSession,
4705     CK_OBJECT_HANDLE_PTR phObject,
4706     CK_ULONG ulMaxObjectCount,
4707     CK_ULONG_PTR pulObjectCount
4708 );
```

4709 **C_FindObjects** continues a search for token and session objects that match a template, obtaining
 4710 additional object handles. *hSession* is the session's handle; *phObject* points to the location that receives
 4711 the list (array) of additional object handles; *ulMaxObjectCount* is the maximum number of object handles
 4712 to be returned; *pulObjectCount* points to the location that receives the actual number of object handles
 4713 returned.

4714 If there are no more objects matching the template, then the location that *pulObjectCount* points to
 4715 receives the value 0.

4716 The search MUST have been initialized with **C_FindObjectsInit**.

4717 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 4718 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4719 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
 4720 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
 4721 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4722 Example: see **C_FindObjectsFinal**.

4723 5.7.9 C_FindObjectsFinal

```
4724 CK_DECLARE_FUNCTION(CK_RV, C_FindObjectsFinal)(
4725     CK_SESSION_HANDLE hSession
4726 );
```

4727 **C_FindObjectsFinal** terminates a search for token and session objects. *hSession* is the session's
 4728 handle.

4729 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 4730 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4731 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
 4732 CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4733 Example:

```
4734 CK_SESSION_HANDLE hSession;
4735 CK_OBJECT_HANDLE hObject;
4736 CK_ULONG ulObjectCount;
4737 CK_RV rv;
4738
4739 .
4740 .
```

```

4741 rv = C_FindObjectsInit(hSession, NULL_PTR, 0);
4742 assert(rv == CKR_OK);
4743 while (1) {
4744     rv = C_FindObjects(hSession, &hObject, 1, &ulObjectCount);
4745     if (rv != CKR_OK || ulObjectCount == 0)
4746         break;
4747     .
4748     .
4749 }
4750
4751 rv = C_FindObjectsFinal(hSession);
4752 assert(rv == CKR_OK);

```

5.8 Encryption functions

Cryptoki provides the following functions for encrypting data:

5.8.1 C_EncryptInit

```

4756 CK_DECLARE_FUNCTION(CK_RV, C_EncryptInit)(
4757     CK_SESSION_HANDLE hSession,
4758     CK_MECHANISM_PTR pMechanism,
4759     CK_OBJECT_HANDLE hKey
4760 );

```

C_EncryptInit initializes an encryption operation. *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_EncryptInit**, the application can either call **C_Encrypt** to encrypt data in a single part; or call **C_EncryptUpdate** zero or more times, followed by **C_EncryptFinal**, to encrypt data in multiple parts. The encryption operation is active until the application uses a call to **C_Encrypt** or **C_EncryptFinal** to *actually obtain* the final piece of ciphertext. To process additional data (in single or multiple parts), the application MUST call **C_EncryptInit** again.

C_EncryptInit can be called with *pMechanism* set to NULL_PTR to terminate an active encryption operation. If an active operation operations cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be returned.

Return values: 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_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

Example: see **C_EncryptFinal**.

5.8.2 C_Encrypt

```

4783 CK_DECLARE_FUNCTION(CK_RV, C_Encrypt)(
4784     CK_SESSION_HANDLE hSession,
4785     CK_BYTE_PTR pData,

```

```

4786     CK_ULONG ulDataLen,
4787     CK_BYTE_PTR pEncryptedData,
4788     CK_ULONG_PTR pulEncryptedDataLen
4789 );

```

4790 **C_Encrypt** encrypts single-part data. *hSession* is the session's handle; *pData* points to the data;
 4791 *ulDataLen* is the length in bytes of the data; *pEncryptedData* points to the location that receives the
 4792 encrypted data; *pulEncryptedDataLen* points to the location that holds the length in bytes of the encrypted
 4793 data.

4794 **C_Encrypt** uses the convention described in Section 5.2 on producing output.

4795 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_Encrypt** always
 4796 terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
 4797 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
 4798 ciphertext.

4799 **C_Encrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_EncryptInit**
 4800 without intervening **C_EncryptUpdate** calls.

4801 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
 4802 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
 4803 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then **C_Encrypt**
 4804 will fail with return code CKR_DATA_LEN_RANGE.

4805 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pData* and *pEncryptedData* point to
 4806 the same location.

4807 For most mechanisms, **C_Encrypt** is equivalent to a sequence of **C_EncryptUpdate** operations followed
 4808 by **C_EncryptFinal**.

4809 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4810 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
 4811 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 4812 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 4813 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
 4814 CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4815 Example: see **C_EncryptFinal** for an example of similar functions.

4816 5.8.3 C_EncryptUpdate

```

4817 CK_DECLARE_FUNCTION(CK_RV, C_EncryptUpdate) (
4818     CK_SESSION_HANDLE hSession,
4819     CK_BYTE_PTR pPart,
4820     CK_ULONG ulPartLen,
4821     CK_BYTE_PTR pEncryptedPart,
4822     CK_ULONG_PTR pulEncryptedPartLen
4823 );

```

4824 **C_EncryptUpdate** continues a multiple-part encryption operation, processing another data part.
 4825 *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part;
 4826 *pEncryptedPart* points to the location that receives the encrypted data part; *pulEncryptedPartLen* points
 4827 to the location that holds the length in bytes of the encrypted data part.

4828 **C_EncryptUpdate** uses the convention described in Section 5.2 on producing output.

4829 The encryption operation MUST have been initialized with **C_EncryptInit**. This function may be called
 4830 any number of times in succession. A call to **C_EncryptUpdate** which results in an error other than
 4831 CKR_BUFFER_TOO_SMALL terminates the current encryption operation.

4832 The plaintext and ciphertext can be in the same place, *i.e.*, it is OK if *pPart* and *pEncryptedPart* point to
 4833 the same location.

4834 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 4835 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,

4836 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4837 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4838 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
4839 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4840 Example: see **C_EncryptFinal**.

4841 5.8.4 C_EncryptFinal

```
4842 CK_DECLARE_FUNCTION(CK_RV, C_EncryptFinal)(  
4843     CK_SESSION_HANDLE hSession,  
4844     CK_BYTE_PTR pLastEncryptedPart,  
4845     CK_ULONG_PTR pulLastEncryptedPartLen  
4846 );
```

4847 **C_EncryptFinal** finishes a multiple-part encryption operation. *hSession* is the session's handle;
4848 *pLastEncryptedPart* points to the location that receives the last encrypted data part, if any;
4849 *pulLastEncryptedPartLen* points to the location that holds the length of the last encrypted data part.

4850 **C_EncryptFinal** uses the convention described in Section 5.2 on producing output.

4851 The encryption operation MUST have been initialized with **C_EncryptInit**. A call to **C_EncryptFinal**
4852 always terminates the active encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
4853 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
4854 ciphertext.

4855 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
4856 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
4857 are not satisfied, then **C_EncryptFinal** will fail with return code CKR_DATA_LEN_RANGE.

4858 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4859 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
4860 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
4861 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
4862 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
4863 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

4864 Example:

```
4865 #define PLAINTEXT_BUF_SZ 200  
4866 #define CIPHERTEXT_BUF_SZ 256  
4867  
4868 CK_ULONG firstPieceLen, secondPieceLen;  
4869 CK_SESSION_HANDLE hSession;  
4870 CK_OBJECT_HANDLE hKey;  
4871 CK_BYTE iv[8];  
4872 CK_MECHANISM mechanism = {  
4873     CKM_DES_CBC_PAD, iv, sizeof(iv)  
4874 };  
4875 CK_BYTE data[PLAINTEXT_BUF_SZ];  
4876 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];  
4877 CK_ULONG ulEncryptedData1Len;  
4878 CK_ULONG ulEncryptedData2Len;  
4879 CK_ULONG ulEncryptedData3Len;  
4880 CK_RV rv;  
4881
```

```

4882 .
4883 .
4884 firstPieceLen = 90;
4885 secondPieceLen = PLAINTEXT_BUF_SZ-firstPieceLen;
4886 rv = C_EncryptInit(hSession, &mechanism, hKey);
4887 if (rv == CKR_OK) {
4888     /* Encrypt first piece */
4889     ulEncryptedData1Len = sizeof(encryptedData);
4890     rv = C_EncryptUpdate(
4891         hSession,
4892         &data[0], firstPieceLen,
4893         &encryptedData[0], &ulEncryptedData1Len);
4894     if (rv != CKR_OK) {
4895         .
4896         .
4897     }
4898
4899     /* Encrypt second piece */
4900     ulEncryptedData2Len = sizeof(encryptedData)-ulEncryptedData1Len;
4901     rv = C_EncryptUpdate(
4902         hSession,
4903         &data[firstPieceLen], secondPieceLen,
4904         &encryptedData[ulEncryptedData1Len], &ulEncryptedData2Len);
4905     if (rv != CKR_OK) {
4906         .
4907         .
4908     }
4909
4910     /* Get last little encrypted bit */
4911     ulEncryptedData3Len =
4912         sizeof(encryptedData)-ulEncryptedData1Len-ulEncryptedData2Len;
4913     rv = C_EncryptFinal(
4914         hSession,
4915         &encryptedData[ulEncryptedData1Len+ulEncryptedData2Len],
4916         &ulEncryptedData3Len);
4917     if (rv != CKR_OK) {
4918         .
4919         .
4920     }
4921 }

```

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.

Return values: **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_PENDING**, **CKR_PIN_EXPIRED**, **CKR_SESSION_CLOSED**, **CKR_SESSION_HANDLE_INVALID**, **CKR_USER_NOT_LOGGED_IN**, **CKR_OPERATION_CANCEL_FAILED**.

5.9.2 C_EncryptMessage

```
CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessage) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen,
    CK_BYTE_PTR pPlaintext,
    CK_ULONG ulPlaintextLen,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG_PTR pulCiphertextLen
);
```

C_EncryptMessage encrypts a message in a single part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pPlaintext* points to the plaintext data; *ulPlaintextLen* is the length in bytes of the plaintext data;

4971 *pCiphertext* points to the location that receives the encrypted data; *pulCiphertextLen* points to the location
4972 that holds the length in bytes of the encrypted data.

4973 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
4974 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
4975 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
4976 generator will be output to the *pParameter* buffer.

4977 If the encryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
4978 should be set to (NULL, 0).

4979 **C_EncryptMessage** uses the convention described in Section 5.2 on producing output.

4980 The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**. A call
4981 to **C_EncryptMessage** begins and terminates a message encryption operation.

4982 **C_EncryptMessage** cannot be called in the middle of a multi-part message encryption operation.

4983 For some encryption mechanisms, the input plaintext data has certain length constraints (either because
4984 the mechanism can only encrypt relatively short pieces of plaintext, or because the mechanism's input
4985 data MUST consist of an integral number of blocks). If these constraints are not satisfied, then
4986 **C_EncryptMessage** will fail with return code CKR_DATA_LEN_RANGE. The plaintext and ciphertext can
4987 be in the same place, i.e., it is OK if *pPlaintext* and *pCiphertext* point to the same location.

4988 For most mechanisms, **C_EncryptMessage** is equivalent to **C_EncryptMessageBegin** followed by a
4989 sequence of **C_EncryptMessageNext** operations.

4990 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
4991 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
4992 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
4993 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
4994 CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE,
4995 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
4996 CKR_SESSION_HANDLE_INVALID.

4997 5.9.3 C_EncryptMessageBegin

```

4998 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageBegin) (
4999     CK_SESSION_HANDLE hSession,
5000     CK_VOID_PTR pParameter,
5001     CK_ULONG ulParameterLen,
5002     CK_BYTE_PTR pAssociatedData,
5003     CK_ULONG ulAssociatedDataLen
5004 );

```

5005 **C_EncryptMessageBegin** begins a multiple-part message encryption operation. *hSession* is the
5006 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5007 message encryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
5008 for an AEAD mechanism.

5009 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
5010 passed to **C_MessageEncryptInit**, *pParameter* may be either an input or an output parameter. For
5011 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
5012 generator will be output to the *pParameter* buffer.

5013 If the mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be
5014 set to (NULL, 0).

5015 After calling **C_EncryptMessageBegin**, the application should call **C_EncryptMessageNext** one or
5016 more times to encrypt the message in multiple parts. The message encryption operation is active until the
5017 application uses a call to **C_EncryptMessageNext** with flags=**CKF_END_OF_MESSAGE** to actually
5018 obtain the final piece of ciphertext. To process additional messages (in single or multiple parts), the
5019 application MUST call **C_EncryptMessage** or **C_EncryptMessageBegin** again.

5020 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5021 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5022 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
5023 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
5024 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5025 CKR_USER_NOT_LOGGED_IN.

5026 5.9.4 C_EncryptMessageNext

```
5027 CK_DECLARE_FUNCTION(CK_RV, C_EncryptMessageNext) (  
5028     CK_SESSION_HANDLE hSession,  
5029     CK_VOID_PTR pParameter,  
5030     CK_ULONG ulParameterLen,  
5031     CK_BYTE_PTR pPlaintextPart,  
5032     CK_ULONG ulPlaintextPartLen,  
5033     CK_BYTE_PTR pCiphertextPart,  
5034     CK_ULONG_PTR pulCiphertextPartLen,  
5035     CK_FLAGS flags  
5036 );
```

5037 **C_EncryptMessageNext** continues a multiple-part message encryption operation, processing another
5038 message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
5039 mechanism-specific parameters for the message encryption operation; *pPlaintextPart* points to the
5040 plaintext message part; *ulPlaintextPartLen* is the length of the plaintext message part; *pCiphertextPart*
5041 points to the location that receives the encrypted message part; *pulCiphertextPartLen* points to the
5042 location that holds the length in bytes of the encrypted message part; flags is set to 0 if there is more
5043 plaintext data to follow, or set to **CKF_END_OF_MESSAGE** if this is the last plaintext part.

5044 Typically, *pParameter* is an initialization vector (IV) or nonce. Depending on the mechanism parameter
5045 passed to **C_EncryptMessageNext**, *pParameter* may be either an input or an output parameter. For
5046 example, if the mechanism parameter specifies an IV generator mechanism, the IV generated by the IV
5047 generator will be output to the *pParameter* buffer.

5048 **C_EncryptMessageNext** uses the convention described in Section 5.2 on producing output.

5049 The message encryption operation MUST have been started with **C_EncryptMessageBegin**. This
5050 function may be called any number of times in succession. A call to **C_EncryptMessageNext** with
5051 flags=0 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message
5052 encryption operation. A call to **C_EncryptMessageNext** with flags=**CKF_END_OF_MESSAGE** always
5053 terminates the active message encryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
5054 successful call (i.e., one which returns **CKR_OK**) to determine the length of the buffer needed to hold the
5055 ciphertext.

5056 Although the last **C_EncryptMessageNext** call ends the encryption of a message, it does not finish the
5057 message-based encryption process. Additional **C_EncryptMessage** or **C_EncryptMessageBegin** and
5058 **C_EncryptMessageNext** calls may be made on the session.

5059 The plaintext and ciphertext can be in the same place, i.e., it is OK if *pPlaintextPart* and *pCiphertextPart*
5060 point to the same location.

5061 For some multi-part encryption mechanisms, the input plaintext data has certain length constraints,
5062 because the mechanism's input data MUST consist of an integral number of blocks. If these constraints
5063 are not satisfied when the final message part is supplied (i.e., with flags=**CKF_END_OF_MESSAGE**),
5064 then **C_EncryptMessageNext** will fail with return code CKR_DATA_LEN_RANGE.

5065 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5066 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
5067 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5068 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
5069 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE,
5070 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5071 CKR_SESSION_HANDLE_INVALID.

5.9.5 C_MessageEncryptFinal

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

C_MessageEncryptFinal finishes a message-based encryption process. hSession is the session's handle.

The message-based encryption process MUST have been initialized with **C_MessageEncryptInit**.

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_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

Example:

```
#define PLAINTEXT_BUF_SZ 200
#define AUTH_BUF_SZ 100
#define CIPHERTEXT_BUF_SZ 256

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hKey;
CK_BYTE iv[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
CK_BYTE tag[16];
CK_GCM_MESSAGE_PARAMS gcmParams = {
    iv,
    sizeof(iv) * 8,
    0,
    CKG_NO_GENERATE,
    tag,
    sizeof(tag) * 8
};
CK_MECHANISM mechanism = {
    CKM_AES_GCM, &gcmParams, sizeof(gcmParams)
};
CK_BYTE data[2][PLAINTEXT_BUF_SZ];
CK_BYTE auth[2][AUTH_BUF_SZ];
CK_BYTE encryptedData[2][CIPHERTEXT_BUF_SZ];
CK_ULONG ulEncryptedDataLen, ulFirstEncryptedDataLen;
CK_ULONG firstPieceLen = PLAINTEXT_BUF_SZ / 2;

/* error handling is omitted for better readability */
.
.
C_MessageEncryptInit(hSession, &mechanism, hKey);
/* encrypt message en bloc with given IV */
ulEncryptedDataLen = sizeof(encryptedData[0]);
```

```

5116 C_EncryptMessage(hSession,
5117     &gcmParams, sizeof(gcmParams),
5118     &auth[0][0], sizeof(auth[0]),
5119     &data[0][0], sizeof(data[0]),
5120     &encryptedData[0][0], &ulEncryptedDataLen);
5121 /* iv and tag are set now for message */
5122
5123 /* encrypt message in two steps with generated IV */
5124 gcmParams.ivGenerator = CKG_GENERATE;
5125 C_EncryptMessageBegin(hSession,
5126     &gcmParams, sizeof(gcmParams),
5127     &auth[1][0], sizeof(auth[1])
5128 );
5129 /* encrypt first piece */
5130 ulFirstEncryptedDataLen = sizeof(encryptedData[1]);
5131 C_EncryptMessageNext(hSession,
5132     &gcmParams, sizeof(gcmParams),
5133     &data[1][0], firstPieceLen,
5134     &encryptedData[1][0], &ulFirstEncryptedDataLen,
5135     0
5136 );
5137 /* encrypt second piece */
5138 ulEncryptedDataLen = sizeof(encryptedData[1]) - ulFirstEncryptedDataLen;
5139 C_EncryptMessageNext(hSession,
5140     &gcmParams, sizeof(gcmParams),
5141     &data[1][firstPieceLen], sizeof(data[1])-firstPieceLen,
5142     &encryptedData[1][ulFirstEncryptedDataLen], &ulEncryptedDataLen,
5143     CKF_END_OF_MESSAGE
5144 );
5145 /* tag is set now for message */
5146
5147 /* finalize */
5148 C_MessageEncryptFinal(hSession);

```

5149 5.10 Decryption functions

5150 Cryptoki provides the following functions for decrypting data:

5151 5.10.1 C_DecryptInit

```

5152 CK_DECLARE_FUNCTION(CK_RV, C_DecryptInit)(
5153     CK_SESSION_HANDLE hSession,
5154     CK_MECHANISM_PTR pMechanism,
5155     CK_OBJECT_HANDLE hKey
5156 );

```

5157 **C_DecryptInit** initializes a decryption operation. *hSession* is the session's handle; *pMechanism* points to
5158 the decryption mechanism; *hKey* is the handle of the decryption key.

5159 The **CKA_DECRYPT** attribute of the decryption key, which indicates whether the key supports
5160 decryption, MUST be CK_TRUE.

5161 After calling **C_DecryptInit**, the application can either call **C_Decrypt** to decrypt data in a single part; or
5162 call **C_DecryptUpdate** zero or more times, followed by **C_DecryptFinal**, to decrypt data in multiple parts.
5163 The decryption operation is active until the application uses a call to **C_Decrypt** or **C_DecryptFinal** to
5164 *actually obtain* the final piece of plaintext. To process additional data (in single or multiple parts), the
5165 application MUST call **C_DecryptInit** again.

5166 **C_DecryptInit** can be called with *pMechanism* set to NULL_PTR to terminate an active decryption
5167 operation. If an active operation cannot be cancelled, CKR_OPERATION_CANCEL_FAILED must be
5168 returned.

5169 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5170 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5171 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5172 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5173 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5174 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
5175 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5176 CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5177 Example: see **C_DecryptFinal**.

5178 5.10.2 C_Decrypt

```
5179 CK_DECLARE_FUNCTION(CK_RV, C_Decrypt) (  
5180     CK_SESSION_HANDLE hSession,  
5181     CK_BYTE_PTR pEncryptedData,  
5182     CK_ULONG ulEncryptedDataLen,  
5183     CK_BYTE_PTR pData,  
5184     CK_ULONG_PTR pulDataLen  
5185 );
```

5186 **C_Decrypt** decrypts encrypted data in a single part. *hSession* is the session's handle; *pEncryptedData*
5187 points to the encrypted data; *ulEncryptedDataLen* is the length of the encrypted data; *pData* points to the
5188 location that receives the recovered data; *pulDataLen* points to the location that holds the length of the
5189 recovered data.

5190 **C_Decrypt** uses the convention described in Section 5.2 on producing output.

5191 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_Decrypt** always
5192 terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
5193 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
5194 plaintext.

5195 **C_Decrypt** cannot be used to terminate a multi-part operation, and MUST be called after **C_DecryptInit**
5196 without intervening **C_DecryptUpdate** calls.

5197 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedData* and *pData* point to
5198 the same location.

5199 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
5200 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

5201 For most mechanisms, **C_Decrypt** is equivalent to a sequence of **C_DecryptUpdate** operations followed
5202 by **C_DecryptFinal**.

5203 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5204 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5205 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
5206 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,

5207 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
5208 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5209 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5210 Example: see **C_DecryptFinal** for an example of similar functions.

5211 5.10.3 C_DecryptUpdate

```
5212 CK_DECLARE_FUNCTION(CK_RV, C_DecryptUpdate) (  
5213     CK_SESSION_HANDLE hSession,  
5214     CK_BYTE_PTR pEncryptedPart,  
5215     CK_ULONG ulEncryptedPartLen,  
5216     CK_BYTE_PTR pPart,  
5217     CK_ULONG_PTR pulPartLen  
5218 );
```

5219 **C_DecryptUpdate** continues a multiple-part decryption operation, processing another encrypted data
5220 part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part;
5221 *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the
5222 recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

5223 **C_DecryptUpdate** uses the convention described in Section 5.2 on producing output.

5224 The decryption operation MUST have been initialized with **C_DecryptInit**. This function may be called
5225 any number of times in succession. A call to **C_DecryptUpdate** which results in an error other than
5226 CKR_BUFFER_TOO_SMALL terminates the current decryption operation.

5227 The ciphertext and plaintext can be in the same place, *i.e.*, it is OK if *pEncryptedPart* and *pPart* point to
5228 the same location.

5229 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5230 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5231 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
5232 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5233 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
5234 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5235 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5236 Example: See **C_DecryptFinal**.

5237 5.10.4 C_DecryptFinal

```
5238 CK_DECLARE_FUNCTION(CK_RV, C_DecryptFinal) (  
5239     CK_SESSION_HANDLE hSession,  
5240     CK_BYTE_PTR pLastPart,  
5241     CK_ULONG_PTR pulLastPartLen  
5242 );
```

5243 **C_DecryptFinal** finishes a multiple-part decryption operation. *hSession* is the session's handle;
5244 *pLastPart* points to the location that receives the last recovered data part, if any; *pulLastPartLen* points to
5245 the location that holds the length of the last recovered data part.

5246 **C_DecryptFinal** uses the convention described in Section 5.2 on producing output.

5247 The decryption operation MUST have been initialized with **C_DecryptInit**. A call to **C_DecryptFinal**
5248 always terminates the active decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
5249 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
5250 plaintext.

5251 If the input ciphertext data cannot be decrypted because it has an inappropriate length, then either
5252 CKR_ENCRYPTED_DATA_INVALID or CKR_ENCRYPTED_DATA_LEN_RANGE may be returned.

5253 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5254 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5255 CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,

5256 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5257 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
5258 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5259 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5260 Example:

```
5261 #define CIPHERTEXT_BUF_SZ 256
5262 #define PLAINTEXT_BUF_SZ 256
5263
5264 CK_ULONG firstEncryptedPieceLen, secondEncryptedPieceLen;
5265 CK_SESSION_HANDLE hSession;
5266 CK_OBJECT_HANDLE hKey;
5267 CK_BYTE iv[8];
5268 CK_MECHANISM mechanism = {
5269     CKM_DES_CBC_PAD, iv, sizeof(iv)
5270 };
5271 CK_BYTE data[PLAINTEXT_BUF_SZ];
5272 CK_BYTE encryptedData[CIPHERTEXT_BUF_SZ];
5273 CK_ULONG ulData1Len, ulData2Len, ulData3Len;
5274 CK_RV rv;
5275
5276 .
5277 .
5278 firstEncryptedPieceLen = 90;
5279 secondEncryptedPieceLen = CIPHERTEXT_BUF_SZ-firstEncryptedPieceLen;
5280 rv = C_DecryptInit(hSession, &mechanism, hKey);
5281 if (rv == CKR_OK) {
5282     /* Decrypt first piece */
5283     ulData1Len = sizeof(data);
5284     rv = C_DecryptUpdate(
5285         hSession,
5286         &encryptedData[0], firstEncryptedPieceLen,
5287         &data[0], &ulData1Len);
5288     if (rv != CKR_OK) {
5289         .
5290         .
5291     }
5292
5293     /* Decrypt second piece */
5294     ulData2Len = sizeof(data)-ulData1Len;
5295     rv = C_DecryptUpdate(
5296         hSession,
5297         &encryptedData[firstEncryptedPieceLen],
5298         secondEncryptedPieceLen,
```

```

5299     &data[ulData1Len], &ulData2Len);
5300     if (rv != CKR_OK) {
5301         .
5302         .
5303     }
5304
5305     /* Get last little decrypted bit */
5306     ulData3Len = sizeof(data)-ulData1Len-ulData2Len;
5307     rv = C_DecryptFinal(
5308         hSession,
5309         &data[ulData1Len+ulData2Len], &ulData3Len);
5310     if (rv != CKR_OK) {
5311         .
5312         .
5313     }
5314 }

```

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

```

5321 CK_DECLARE_FUNCTION(CK_RV, C_MessageDecryptInit)(
5322     CK_SESSION_HANDLE hSession,
5323     CK_MECHANISM_PTR pMechanism,
5324     CK_OBJECT_HANDLE hKey
5325 );

```

C_MessageDecryptInit initializes a message-based decryption process, preparing a session for one or more decryption operations that use the same decryption mechanism and decryption key. *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_MessageDecryptInit**, the application can either call **C_DecryptMessage** to decrypt an encrypted message in a single part; or call **C_DecryptMessageBegin**, followed by **C_DecryptMessageNext** one or more times, to decrypt an encrypted message in multiple parts. This may be repeated several times. The message-based decryption process is active until the application uses a call to **C_MessageDecryptFinal** to finish the message-based decryption process.

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_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5.11.2 C_DecryptMessage

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessage) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG ulCiphertextLen,
    CK_BYTE_PTR pPlaintext,
    CK_ULONG_PTR pulPlaintextLen
);
```

C_DecryptMessage decrypts an encrypted message in a single part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pCiphertext* points to the encrypted message; *ulCiphertextLen* is the length of the encrypted message; *pPlaintext* points to the location that receives the recovered message; *pulPlaintextLen* points to the location that holds the length of the recovered message.

Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of **C_EncryptMessage**, *pParameter* is always an input parameter.

If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and should be set to (NULL, 0).

C_DecryptMessage uses the convention described in Section 5.2 on producing output.

The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**. A call to **C_DecryptMessage** begins and terminates a message decryption operation.

C_DecryptMessage cannot be called in the middle of a multi-part message decryption operation.

The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertext* and *pPlaintext* point to the same location.

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.

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned.

For most mechanisms, **C_DecryptMessage** is equivalent to **C_DecryptMessageBegin** followed by a sequence of **C_DecryptMessageNext** operations.

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_AEAD_DECRYPT_FAILED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5.11.3 C_DecryptMessageBegin

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageBegin) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pAssociatedData,
    CK_ULONG ulAssociatedDataLen
```

5395);

5396 **C_DecryptMessageBegin** begins a multiple-part message decryption operation. *hSession* is the
5397 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
5398 message decryption operation; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data
5399 for an AEAD mechanism.

5400 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
5401 **C_EncryptMessageBegin**, *pParameter* is always an input parameter.

5402 If the decryption mechanism is not AEAD, *pAssociatedData* and *ulAssociatedDataLen* are not used and
5403 should be set to (NULL, 0).

5404 After calling **C_DecryptMessageBegin**, the application should call **C_DecryptMessageNext** one or
5405 more times to decrypt the encrypted message in multiple parts. The message decryption operation is
5406 active until the application uses a call to **C_DecryptMessageNext** with flags=**CKF_END_OF_MESSAGE**
5407 to actually obtain the final piece of plaintext. To process additional encrypted messages (in single or
5408 multiple parts), the application MUST call **C_DecryptMessage** or **C_DecryptMessageBegin** again.

5409 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5410 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5411 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5412 CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE,
5413 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_PIN_EXPIRED,
5414 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5415 5.11.4 C_DecryptMessageNext

```
5416 CK_DECLARE_FUNCTION(CK_RV, C_DecryptMessageNext) (  
5417     CK_SESSION_HANDLE hSession,  
5418     CK_VOID_PTR pParameter,  
5419     CK_ULONG ulParameterLen,  
5420     CK_BYTE_PTR pCiphertextPart,  
5421     CK_ULONG ulCiphertextPartLen,  
5422     CK_BYTE_PTR pPlaintextPart,  
5423     CK_ULONG_PTR pulPlaintextPartLen,  
5424     CK_FLAGS flags  
5425 );
```

5426 **C_DecryptMessageNext** continues a multiple-part message decryption operation, processing another
5427 encrypted message part. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any
5428 mechanism-specific parameters for the message decryption operation; *pCiphertextPart* points to the
5429 encrypted message part; *ulCiphertextPartLen* is the length of the encrypted message part; *pPlaintextPart*
5430 points to the location that receives the recovered message part; *pulPlaintextPartLen* points to the location
5431 that holds the length of the recovered message part; flags is set to 0 if there is more ciphertext data to
5432 follow, or set to **CKF_END_OF_MESSAGE** if this is the last ciphertext part.

5433 Typically, *pParameter* is an initialization vector (IV) or nonce. Unlike the *pParameter* parameter of
5434 **C_EncryptMessageNext**, *pParameter* is always an input parameter.

5435 **C_DecryptMessageNext** uses the convention described in Section 5.2 on producing output.

5436 The message decryption operation MUST have been started with **C_DecryptMessageBegin**. This
5437 function may be called any number of times in succession. A call to **C_DecryptMessageNext** with
5438 flags=0 which results in an error other than CKR_BUFFER_TOO_SMALL terminates the current message
5439 decryption operation. A call to **C_DecryptMessageNext** with flags=**CKF_END_OF_MESSAGE** always
5440 terminates the active message decryption operation unless it returns CKR_BUFFER_TOO_SMALL or is a
5441 successful call (i.e., one which returns CKR_OK) to determine the length of the buffer needed to hold the
5442 plaintext.

5443 The ciphertext and plaintext can be in the same place, i.e., it is OK if *pCiphertextPart* and *pPlaintextPart*
5444 point to the same location.

Although the last **C_DecryptMessageNext** call ends the decryption of a message, it does not finish the message-based decryption process. Additional **C_DecryptMessage** or **C_DecryptMessageBegin** and **C_DecryptMessageNext** calls may be made on the session.

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 by the last **C_DecryptMessageNext** call.

If the decryption mechanism is an AEAD algorithm and the authenticity of the associated data or ciphertext cannot be verified, then CKR_AEAD_DECRYPT_FAILED is returned by the last **C_DecryptMessageNext** call.

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_AEAD_DECRYPT_FAILED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.11.5 C_MessageDecryptFinal

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

C_MessageDecryptFinal finishes a message-based decryption process. *hSession* is the session's handle.

The message-based decryption process MUST have been initialized with **C_MessageDecryptInit**.

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_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.12 Message digesting functions

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.

5492 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5493 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5494 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5495 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID,
5496 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_PIN_EXPIRED,
5497 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5498 CKR_OPERATION_CANCEL_FAILED.

5499 Example: see **C_DigestFinal**.

5500 5.12.2 C_Digest

```
5501 CK_DECLARE_FUNCTION(CK_RV, C_Digest) (  
5502     CK_SESSION_HANDLE hSession,  
5503     CK_BYTE_PTR pData,  
5504     CK_ULONG ulDataLen,  
5505     CK_BYTE_PTR pDigest,  
5506     CK_ULONG_PTR pulDigestLen  
5507 );
```

5508 **C_Digest** digests data in a single part. *hSession* is the session's handle, *pData* points to the data;
5509 *ulDataLen* is the length of the data; *pDigest* points to the location that receives the message digest;
5510 *pulDigestLen* points to the location that holds the length of the message digest.

5511 **C_Digest** uses the convention described in Section 5.2 on producing output.

5512 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_Digest** always terminates
5513 the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one
5514 which returns CKR_OK) to determine the length of the buffer needed to hold the message digest.

5515 **C_Digest** cannot be used to terminate a multi-part operation, and MUST be called after **C_DigestInit**
5516 without intervening **C_DigestUpdate** calls.

5517 The input data and digest output can be in the same place, *i.e.*, it is OK if *pData* and *pDigest* point to the
5518 same location.

5519 **C_Digest** is equivalent to a sequence of **C_DigestUpdate** operations followed by **C_DigestFinal**.

5520 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5521 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5522 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5523 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
5524 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5525 CKR_SESSION_HANDLE_INVALID.

5526 Example: see **C_DigestFinal** for an example of similar functions.

5527 5.12.3 C_DigestUpdate

```
5528 CK_DECLARE_FUNCTION(CK_RV, C_DigestUpdate) (  
5529     CK_SESSION_HANDLE hSession,  
5530     CK_BYTE_PTR pPart,  
5531     CK_ULONG ulPartLen  
5532 );
```

5533 **C_DigestUpdate** continues a multiple-part message-digesting operation, processing another data part.
5534 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

5535 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
5536 and **C_DigestKey** may be interspersed any number of times in any order. A call to **C_DigestUpdate**
5537 which results in an error terminates the current digest operation.

5538 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5539 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5540 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,

5541 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
5542 CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

5543 Example: see **C_DigestFinal**.

5544 5.12.4 C_DigestKey

```
5545 CK_DECLARE_FUNCTION(CK_RV, C_DigestKey) (  
5546     CK_SESSION_HANDLE hSession,  
5547     CK_OBJECT_HANDLE hKey  
5548 );
```

5549 **C_DigestKey** continues a multiple-part message-digesting operation by digesting the value of a secret
5550 key. *hSession* is the session's handle; *hKey* is the handle of the secret key to be digested.

5551 The message-digesting operation MUST have been initialized with **C_DigestInit**. Calls to this function
5552 and **C_DigestUpdate** may be interspersed any number of times in any order.

5553 If the value of the supplied key cannot be digested purely for some reason related to its length,
5554 **C_DigestKey** should return the error code CKR_KEY_SIZE_RANGE.

5555 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5556 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5557 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
5558 CKR_KEY_INDIGESTIBLE, CKR_KEY_SIZE_RANGE, CKR_OK, CKR_OPERATION_ACTIVE,
5559 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5560 CKR_SESSION_HANDLE_INVALID.

5561 Example: see **C_DigestFinal**.

5562 5.12.5 C_DigestFinal

```
5563 CK_DECLARE_FUNCTION(CK_RV, C_DigestFinal) (  
5564     CK_SESSION_HANDLE hSession,  
5565     CK_BYTE_PTR pDigest,  
5566     CK_ULONG_PTR pulDigestLen  
5567 );
```

5568 **C_DigestFinal** finishes a multiple-part message-digesting operation, returning the message digest.
5569 *hSession* is the session's handle; *pDigest* points to the location that receives the message digest;
5570 *pulDigestLen* points to the location that holds the length of the message digest.

5571 **C_DigestFinal** uses the convention described in Section 5.2 on producing output.

5572 The digest operation MUST have been initialized with **C_DigestInit**. A call to **C_DigestFinal** always
5573 terminates the active digest operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
5574 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the message
5575 digest.

5576 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5577 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5578 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5579 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
5580 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5581 CKR_SESSION_HANDLE_INVALID.

5582 Example:

```
5583 CK_SESSION_HANDLE hSession;  
5584 CK_OBJECT_HANDLE hKey;  
5585 CK_MECHANISM mechanism = {  
5586     CKM_MD5, NULL_PTR, 0  
5587 };
```

```

5588 CK_BYTE data[] = {...};
5589 CK_BYTE digest[16];
5590 CK_ULONG ulDigestLen;
5591 CK_RV rv;
5592
5593 .
5594 .
5595 rv = C_DigestInit(hSession, &mechanism);
5596 if (rv != CKR_OK) {
5597     .
5598     .
5599 }
5600
5601 rv = C_DigestUpdate(hSession, data, sizeof(data));
5602 if (rv != CKR_OK) {
5603     .
5604     .
5605 }
5606
5607 rv = C_DigestKey(hSession, hKey);
5608 if (rv != CKR_OK) {
5609     .
5610     .
5611 }
5612
5613 ulDigestLen = sizeof(digest);
5614 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
5615 .
5616 .

```

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

```

5621 CK_DECLARE_FUNCTION(CK_RV, C_SignInit)(
5622     CK_SESSION_HANDLE hSession,
5623     CK_MECHANISM_PTR pMechanism,
5624     CK_OBJECT_HANDLE hKey
5625 );

```

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.

5631 After calling **C_SignInit**, the application can either call **C_Sign** to sign in a single part; or call
5632 **C_SignUpdate** one or more times, followed by **C_SignFinal**, to sign data in multiple parts. The signature
5633 operation is active until the application uses a call to **C_Sign** or **C_SignFinal** to *actually obtain* the
5634 signature. To process additional data (in single or multiple parts), the application MUST call **C_SignInit**
5635 again.

5636 **C_SignInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature operation.
5637 If an operation has been initialized and it cannot be cancelled, CKR_OPERATION_CANCEL_FAILED
5638 must be returned.

5639 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5640 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5641 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5642 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5643 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5644 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
5645 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5646 CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5647 Example: see **C_SignFinal**.

5648 5.13.2 C_Sign

```
5649 CK_DECLARE_FUNCTION(CK_RV, C_Sign) (  
5650     CK_SESSION_HANDLE hSession,  
5651     CK_BYTE_PTR pData,  
5652     CK_ULONG ulDataLen,  
5653     CK_BYTE_PTR pSignature,  
5654     CK_ULONG_PTR pulSignatureLen  
5655 );
```

5656 **C_Sign** signs data in a single part, where the signature is an appendix to the data. *hSession* is the
5657 session's handle; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature* points to the
5658 location that receives the signature; *pulSignatureLen* points to the location that holds the length of the
5659 signature.

5660 **C_Sign** uses the convention described in Section 5.2 on producing output.

5661 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_Sign** always terminates
5662 the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*,
5663 one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

5664 **C_Sign** cannot be used to terminate a multi-part operation, and MUST be called after **C_SignInit** without
5665 intervening **C_SignUpdate** calls.

5666 For most mechanisms, **C_Sign** is equivalent to a sequence of **C_SignUpdate** operations followed by
5667 **C_SignFinal**.

5668 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5669 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
5670 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5671 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5672 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
5673 CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5674 CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
5675 CKR_TOKEN_RESOURCE_EXCEEDED.

5676 Example: see **C_SignFinal** for an example of similar functions.

5677 5.13.3 C_SignUpdate

```
5678 CK_DECLARE_FUNCTION(CK_RV, C_SignUpdate) (  
5679     CK_SESSION_HANDLE hSession,
```

```

5680     CK_BYTE_PTR pPart,
5681     CK_ULONG ulPartLen
5682 );

```

5683 **C_SignUpdate** continues a multiple-part signature operation, processing another data part. *hSession* is
5684 the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

5685 The signature operation MUST have been initialized with **C_SignInit**. This function may be called any
5686 number of times in succession. A call to **C_SignUpdate** which results in an error terminates the current
5687 signature operation.

5688 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5689 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
5690 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
5691 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
5692 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5693 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5694 CKR_TOKEN_RESOURCE_EXCEEDED.

5695 Example: see **C_SignFinal**.

5696 5.13.4 C_SignFinal

```

5697 CK_DECLARE_FUNCTION(CK_RV, C_SignFinal)(
5698     CK_SESSION_HANDLE hSession,
5699     CK_BYTE_PTR pSignature,
5700     CK_ULONG_PTR pulSignatureLen
5701 );

```

5702 **C_SignFinal** finishes a multiple-part signature operation, returning the signature. *hSession* is the
5703 session's handle; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to
5704 the location that holds the length of the signature.

5705 **C_SignFinal** uses the convention described in Section 5.2 on producing output.

5706 The signing operation MUST have been initialized with **C_SignInit**. A call to **C_SignFinal** always
5707 terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a successful
5708 call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the signature.

5709 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5710 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
5711 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
5712 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
5713 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
5714 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5715 CKR_FUNCTION_REJECTED, CKR_TOKEN_RESOURCE_EXCEEDED.

5716 Example:

```

5717 CK_SESSION_HANDLE hSession;
5718 CK_OBJECT_HANDLE hKey;
5719 CK_MECHANISM mechanism = {
5720     CKM_DES_MAC, NULL_PTR, 0
5721 };
5722 CK_BYTE data[] = {...};
5723 CK_BYTE mac[4];
5724 CK_ULONG ulMacLen;
5725 CK_RV rv;
5726
5727 .

```

```

5728 .
5729 rv = C_SignInit(hSession, &mechanism, hKey);
5730 if (rv == CKR_OK) {
5731     rv = C_SignUpdate(hSession, data, sizeof(data));
5732     .
5733     .
5734     ulMacLen = sizeof(mac);
5735     rv = C_SignFinal(hSession, mac, &ulMacLen);
5736     .
5737     .
5738 }

```

5739 5.13.5 C_SignRecoverInit

```

5740 CK_DECLARE_FUNCTION(CK_RV, C_SignRecoverInit) (
5741     CK_SESSION_HANDLE hSession,
5742     CK_MECHANISM_PTR pMechanism,
5743     CK_OBJECT_HANDLE hKey
5744 );

```

5745 **C_SignRecoverInit** initializes a signature operation, where the data can be recovered from the signature.
5746 *hSession* is the session's handle; *pMechanism* points to the structure that specifies the signature
5747 mechanism; *hKey* is the handle of the signature key.

5748 The **CKA_SIGN_RECOVER** attribute of the signature key, which indicates whether the key supports
5749 signatures where the data can be recovered from the signature, MUST be CK_TRUE.

5750 After calling **C_SignRecoverInit**, the application may call **C_SignRecover** to sign in a single part. The
5751 signature operation is active until the application uses a call to **C_SignRecover** to *actually obtain* the
5752 signature. To process additional data in a single part, the application MUST call **C_SignRecoverInit**
5753 again.

5754 **C_SignRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active signature
5755 with data recovery operation. If an active operation has been initialized and it cannot be cancelled,
5756 CKR_OPERATION_CANCEL_FAILED must be returned.

5757 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5758 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5759 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5760 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5761 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5762 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
5763 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5764 CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5765 Example: see **C_SignRecover**.

5766 5.13.6 C_SignRecover

```

5767 CK_DECLARE_FUNCTION(CK_RV, C_SignRecover) (
5768     CK_SESSION_HANDLE hSession,
5769     CK_BYTE_PTR pData,
5770     CK_ULONG ulDataLen,
5771     CK_BYTE_PTR pSignature,
5772     CK_ULONG_PTR pulSignatureLen
5773 );

```

5774 **C_SignRecover** signs data in a single operation, where the data can be recovered from the signature.
5775 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
5776 *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that
5777 holds the length of the signature.

5778 **C_SignRecover** uses the convention described in Section 5.2 on producing output.

5779 The signing operation MUST have been initialized with **C_SignRecoverInit**. A call to **C_SignRecover**
5780 always terminates the active signing operation unless it returns CKR_BUFFER_TOO_SMALL or is a
5781 successful call (*i.e.*, one which returns CKR_OK) to determine the length of the buffer needed to hold the
5782 signature.

5783 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5784 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
5785 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5786 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5787 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
5788 CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5789 CKR_USER_NOT_LOGGED_IN, CKR_TOKEN_RESOURCE_EXCEEDED.

5790 Example:

```
5791 CK_SESSION_HANDLE hSession;  
5792 CK_OBJECT_HANDLE hKey;  
5793 CK_MECHANISM mechanism = {  
5794     CKM_RSA_9796, NULL_PTR, 0  
5795 };  
5796 CK_BYTE data[] = {...};  
5797 CK_BYTE signature[128];  
5798 CK_ULONG ulSignatureLen;  
5799 CK_RV rv;  
5800  
5801 .  
5802 .  
5803 rv = C_SignRecoverInit(hSession, &mechanism, hKey);  
5804 if (rv == CKR_OK) {  
5805     ulSignatureLen = sizeof(signature);  
5806     rv = C_SignRecover(  
5807         hSession, data, sizeof(data), signature, &ulSignatureLen);  
5808     if (rv == CKR_OK) {  
5809         .  
5810         .  
5811     }  
5812 }  
5813
```

5814 5.14 Message-based signing and MACing functions

5815 Message-based signature refers to the process of signing multiple messages using the same signature
5816 mechanism and signature key.

5817 Cryptoki provides the following functions for signing messages (for the purposes of Cryptoki, these
5818 operations also encompass message authentication codes).

5.14.1 C_MessageSignInit

```
CK_DECLARE_FUNCTION(CK_RV, C_MessageSignInit) (
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hKey
);
```

C_MessageSignInit initializes a message-based signature process, preparing a session for one or more signature operations (where the signature is an appendix to the data) that use the same signature mechanism and signature key. *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_MessageSignInit**, the application can either call **C_SignMessage** to sign a message in a single part; or call **C_SignMessageBegin**, followed by **C_SignMessageNext** one or more times, to sign a message in multiple parts. This may be repeated several times. The message-based signature process is active until the application calls **C_MessageSignFinal** to finish the message-based signature process.

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_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

5.14.2 C_SignMessage

```
CK_DECLARE_FUNCTION(CK_RV, C_SignMessage) (
    CK_SESSION_HANDLE hSession,
    CK_VOID_PTR pParameter,
    CK_ULONG ulParameterLen,
    CK_BYTE_PTR pData,
    CK_ULONG ulDataLen,
    CK_BYTE_PTR pSignature,
    CK_ULONG_PTR pulSignatureLen
);
```

C_SignMessage signs a message in a single part, where the signature is an appendix to the message. **C_MessageSignInit** must previously been called on the session. *hSession* is the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature operation; *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.

Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an input or an output parameter.

C_SignMessage uses the convention described in Section 5.2 on producing output.

The message-based signing process MUST have been initialized with **C_MessageSignInit**. A call to **C_SignMessage** begins and terminates a message signing operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a successful call (i.e., one which returns CKR_OK).

C_SignMessage cannot be called in the middle of a multi-part message signing operation.

5866 **C_SignMessage** does not finish the message-based signing process. Additional **C_SignMessage** or
5867 **C_SignMessageBegin** and **C_SignMessageNext** calls may be made on the session.

5868 For most mechanisms, **C_SignMessage** is equivalent to **C_SignMessageBegin** followed by a sequence
5869 of **C_SignMessageNext** operations.

5870 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
5871 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
5872 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5873 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5874 CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE,
5875 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
5876 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED,
5877 CKR_TOKEN_RESOURCE_EXCEEDED.

5878 5.14.3 C_SignMessageBegin

```
5879 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageBegin) (
5880     CK_SESSION_HANDLE hSession,
5881     CK_VOID_PTR pParameter,
5882     CK_ULONG ulParameterLen
5883 );
```

5884 **C_SignMessageBegin** begins a multiple-part message signature operation, where the signature is an
5885 appendix to the message. **C_MessageSignInit** must previously been called on the session. *hSession* is
5886 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
5887 the message signature operation.

5888 Depending on the mechanism parameter passed to **C_MessageSignInit**, *pParameter* may be either an
5889 input or an output parameter.

5890 After calling **C_SignMessageBegin**, the application should call **C_SignMessageNext** one or more times
5891 to sign the message in multiple parts. The message signature operation is active until the application
5892 uses a call to **C_SignMessageNext** with a non-NULL *pulSignatureLen* to actually obtain the signature.
5893 To process additional messages (in single or multiple parts), the application MUST call **C_SignMessage**
5894 or **C_SignMessageBegin** again.

5895 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5896 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5897 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5898 CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE,
5899 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_PIN_EXPIRED,
5900 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
5901 CKR_TOKEN_RESOURCE_EXCEEDED.

5902 5.14.4 C_SignMessageNext

```
5903 CK_DECLARE_FUNCTION(CK_RV, C_SignMessageNext) (
5904     CK_SESSION_HANDLE hSession,
5905     CK_VOID_PTR pParameter,
5906     CK_ULONG ulParameterLen,
5907     CK_BYTE_PTR pDataPart,
5908     CK_ULONG ulDataPartLen,
5909     CK_BYTE_PTR pSignature,
5910     CK_ULONG_PTR pulSignatureLen
5911 );
```

C_SignMessageNext continues a multiple-part message signature operation, processing another data part, or finishes a multiple-part message signature operation, returning the signature. *hSession* is the session's handle, *pDataPart* points to the data part; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the message signature operation; *ulDataPartLen* is the length of the data part; *pSignature* points to the location that receives the signature; *pulSignatureLen* points to the location that holds the length of the signature.

The *pulSignatureLen* argument is set to NULL if there is more data part to follow, or set to a non-NULL value (to receive the signature length) if this is the last data part.

C_SignMessageNext uses the convention described in Section 5.2 on producing output.

The message signing operation MUST have been started with **C_SignMessageBegin**. This function may be called any number of times in succession. A call to **C_SignMessageNext** with a NULL *pulSignatureLen* which results in an error terminates the current message signature operation. A call to **C_SignMessageNext** with a non-NULL *pulSignatureLen* always terminates the active message signing operation unless it returns CKR_BUFFER_TOO_SMALL to determine the length of the buffer needed to hold the signature, or is a successful call (i.e., one which returns CKR_OK).

Although the last **C_SignMessageNext** call ends the signing of a message, it does not finish the message-based signing process. Additional **C_SignMessage** or **C_SignMessageBegin** and **C_SignMessageNext** calls may be made on the session.

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_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED, CKR_TOKEN_RESOURCE_EXCEEDED.

5.14.5 C_MessageSignFinal

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

C_MessageSignFinal finishes a message-based signing process. *hSession* is the session's handle.

The message-based signing process MUST have been initialized with **C_MessageSignInit**.

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_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN, CKR_FUNCTION_REJECTED, CKR_TOKEN_RESOURCE_EXCEEDED.

5.15 Functions for verifying signatures and MACs

Cryptoki provides the following functions for verifying signatures on data (for the purposes of Cryptoki, these operations also encompass message authentication codes):

5.15.1 C_VerifyInit

```
CK_DECLARE_FUNCTION(CK_RV, C_VerifyInit) (  
    CK_SESSION_HANDLE hSession,  
    CK_MECHANISM_PTR pMechanism,
```

```

5958     CK_OBJECT_HANDLE hKey
5959 );

```

5960 **C_VerifyInit** initializes a verification operation, where the signature is an appendix to the data. *hSession*
5961 is the session's handle; *pMechanism* points to the structure that specifies the verification mechanism;
5962 *hKey* is the handle of the verification key.

5963 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
5964 where the signature is an appendix to the data, MUST be CK_TRUE.

5965 After calling **C_VerifyInit**, the application can either call **C_Verify** to verify a signature on data in a single
5966 part; or call **C_VerifyUpdate** one or more times, followed by **C_VerifyFinal**, to verify a signature on data
5967 in multiple parts. The verification operation is active until the application calls **C_Verify** or **C_VerifyFinal**.
5968 To process additional data (in single or multiple parts), the application MUST call **C_VerifyInit** again.

5969 **C_VerifyInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
5970 operation. If an active operation has been initialized and it cannot be cancelled,
5971 CKR_OPERATION_CANCEL_FAILED must be returned.

5972 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
5973 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
5974 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
5975 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
5976 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
5977 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
5978 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
5979 CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

5980 Example: see **C_VerifyFinal**.

5981 5.15.2 C_Verify

```

5982 CK_DECLARE_FUNCTION(CK_RV, C_Verify)(
5983     CK_SESSION_HANDLE hSession,
5984     CK_BYTE_PTR pData,
5985     CK_ULONG ulDataLen,
5986     CK_BYTE_PTR pSignature,
5987     CK_ULONG ulSignatureLen
5988 );

```

5989 **C_Verify** verifies a signature in a single-part operation, where the signature is an appendix to the data.
5990 *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data;
5991 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

5992 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_Verify** always
5993 terminates the active verification operation.

5994 A successful call to **C_Verify** should return either the value CKR_OK (indicating that the supplied
5995 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
5996 signature can be seen to be invalid purely on the basis of its length, then
5997 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active signing operation
5998 is terminated.

5999 **C_Verify** cannot be used to terminate a multi-part operation, and MUST be called after **C_VerifyInit**
6000 without intervening **C_VerifyUpdate** calls.

6001 For most mechanisms, **C_Verify** is equivalent to a sequence of **C_VerifyUpdate** operations followed by
6002 **C_VerifyFinal**.

6003 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
6004 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6005 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6006 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6007 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,

6008 CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID, CKR_SIGNATURE_LEN_RANGE,
6009 CKR_TOKEN_RESOURCE_EXCEEDED.

6010 Example: see **C_VerifyFinal** for an example of similar functions.

6011 5.15.3 C_VerifyUpdate

```
6012 CK_DECLARE_FUNCTION(CK_RV, C_VerifyUpdate) (  
6013     CK_SESSION_HANDLE hSession,  
6014     CK_BYTE_PTR pPart,  
6015     CK_ULONG ulPartLen  
6016 );
```

6017 **C_VerifyUpdate** continues a multiple-part verification operation, processing another data part. *hSession*
6018 is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

6019 The verification operation MUST have been initialized with **C_VerifyInit**. This function may be called any
6020 number of times in succession. A call to **C_VerifyUpdate** which results in an error terminates the current
6021 verification operation.

6022 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6023 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6024 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6025 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6026 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
6027 CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_RESOURCE_EXCEEDED.

6028 Example: see **C_VerifyFinal**.

6029 5.15.4 C_VerifyFinal

```
6030 CK_DECLARE_FUNCTION(CK_RV, C_VerifyFinal) (  
6031     CK_SESSION_HANDLE hSession,  
6032     CK_BYTE_PTR pSignature,  
6033     CK_ULONG ulSignatureLen  
6034 );
```

6035 **C_VerifyFinal** finishes a multiple-part verification operation, checking the signature. *hSession* is the
6036 session's handle; *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

6037 The verification operation MUST have been initialized with **C_VerifyInit**. A call to **C_VerifyFinal** always
6038 terminates the active verification operation.

6039 A successful call to **C_VerifyFinal** should return either the value CKR_OK (indicating that the supplied
6040 signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is invalid). If the
6041 signature can be seen to be invalid purely on the basis of its length, then
6042 CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active verifying
6043 operation is terminated.

6044 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6045 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6046 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6047 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6048 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
6049 CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID, CKR_SIGNATURE_LEN_RANGE,
6050 CKR_TOKEN_RESOURCE_EXCEEDED.

6051 Example:

```
6052 CK_SESSION_HANDLE hSession;  
6053 CK_OBJECT_HANDLE hKey;  
6054 CK_MECHANISM mechanism = {  
6055     CKM_DES_MAC, NULL_PTR, 0
```

```

6056 };
6057 CK_BYTE data[] = {...};
6058 CK_BYTE mac[4];
6059 CK_RV rv;
6060
6061 .
6062 .
6063 rv = C_VerifyInit(hSession, &mechanism, hKey);
6064 if (rv == CKR_OK) {
6065     rv = C_VerifyUpdate(hSession, data, sizeof(data));
6066     .
6067     .
6068     rv = C_VerifyFinal(hSession, mac, sizeof(mac));
6069     .
6070     .
6071 }

```

6072 5.15.5 C_VerifyRecoverInit

```

6073 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecoverInit) (
6074     CK_SESSION_HANDLE hSession,
6075     CK_MECHANISM_PTR pMechanism,
6076     CK_OBJECT_HANDLE hKey
6077 );

```

6078 **C_VerifyRecoverInit** initializes a signature verification operation, where the data is recovered from the
6079 signature. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the
6080 verification mechanism; *hKey* is the handle of the verification key.

6081 The **CKA_VERIFY_RECOVER** attribute of the verification key, which indicates whether the key supports
6082 verification where the data is recovered from the signature, **MUST** be CK_TRUE.

6083 After calling **C_VerifyRecoverInit**, the application may call **C_VerifyRecover** to verify a signature on
6084 data in a single part. The verification operation is active until the application uses a call to
6085 **C_VerifyRecover** to *actually obtain* the recovered message.

6086 **C_VerifyRecoverInit** can be called with *pMechanism* set to NULL_PTR to terminate an active verification
6087 with data recovery operation. If an active operations has been initialized and it cannot be cancelled,
6088 CKR_OPERATION_CANCEL_FAILED must be returned.

6089 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6090 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6091 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6092 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
6093 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
6094 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
6095 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
6096 CKR_USER_NOT_LOGGED_IN, CKR_OPERATION_CANCEL_FAILED.

6097 Example: see **C_VerifyRecover**.

6098 5.15.6 C_VerifyRecover

```

6099 CK_DECLARE_FUNCTION(CK_RV, C_VerifyRecover) (
6100     CK_SESSION_HANDLE hSession,
6101     CK_BYTE_PTR pSignature,

```

```

6102     CK_ULONG ulSignatureLen,
6103     CK_BYTE_PTR pData,
6104     CK_ULONG_PTR pulDataLen
6105 );

```

6106 **C_VerifyRecover** verifies a signature in a single-part operation, where the data is recovered from the
6107 signature. *hSession* is the session's handle; *pSignature* points to the signature; *ulSignatureLen* is the
6108 length of the signature; *pData* points to the location that receives the recovered data; and *pulDataLen*
6109 points to the location that holds the length of the recovered data.

6110 **C_VerifyRecover** uses the convention described in Section 5.2 on producing output.

6111 The verification operation MUST have been initialized with **C_VerifyRecoverInit**. A call to
6112 **C_VerifyRecover** always terminates the active verification operation unless it returns
6113 CKR_BUFFER_TOO_SMALL or is a successful call (*i.e.*, one which returns CKR_OK) to determine the
6114 length of the buffer needed to hold the recovered data.

6115 A successful call to **C_VerifyRecover** should return either the value CKR_OK (indicating that the
6116 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
6117 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
6118 CKR_SIGNATURE_LEN_RANGE should be returned. The return codes CKR_SIGNATURE_INVALID
6119 and CKR_SIGNATURE_LEN_RANGE have a higher priority than the return code
6120 CKR_BUFFER_TOO_SMALL, *i.e.*, if **C_VerifyRecover** is supplied with an invalid signature, it will never
6121 return CKR_BUFFER_TOO_SMALL.

6122 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
6123 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID, CKR_DATA_LEN_RANGE,
6124 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6125 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6126 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED,
6127 CKR_PENDING, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
6128 CKR_SIGNATURE_LEN_RANGE, CKR_SIGNATURE_INVALID,
6129 CKR_TOKEN_RESOURCE_EXCEEDED.

6130 Example:

```

6131 CK_SESSION_HANDLE hSession;
6132 CK_OBJECT_HANDLE hKey;
6133 CK_MECHANISM mechanism = {
6134     CKM_RSA_9796, NULL_PTR, 0
6135 };
6136 CK_BYTE data[] = {...};
6137 CK_ULONG ulDataLen;
6138 CK_BYTE signature[128];
6139 CK_RV rv;
6140
6141 .
6142 .
6143 rv = C_VerifyRecoverInit(hSession, &mechanism, hKey);
6144 if (rv == CKR_OK) {
6145     ulDataLen = sizeof(data);
6146     rv = C_VerifyRecover(
6147         hSession, signature, sizeof(signature), data, &ulDataLen);
6148     .
6149     .

```

6150 | }

6151 5.15.7 C_VerifySignatureInit

```
6152 CK_DECLARE_FUNCTION(CK_RV, C_VerifySignatureInit) (  
6153     CK_SESSION_HANDLE hSession,  
6154     CK_MECHANISM_PTR pMechanism,  
6155     CK_OBJECT_HANDLE hKey,  
6156     CK_BYTE_PTR pSignature,  
6157     CK_ULONG ulSignatureLen  
6158 );
```

6159 **C_VerifySignatureInit** initializes a verification operation, where the signature is included as part of the
6160 initialization. *hSession* is the session's handle; *pMechanism* points to the structure that specifies the
6161 verification mechanism; *hKey* is the handle of the verification key; *pSignature* points to the signature;
6162 *ulSignatureLen* is the length of the signature.

6163 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification,
6164 MUST be CK_TRUE.

6165 After calling **C_VerifySignatureInit**, the application can either call **C_VerifySignature** to verify a
6166 signature on data in a single part; or call **C_VerifySignatureUpdate** one or more times, followed by
6167 **C_VerifySignatureFinal**, to verify a signature on data in multiple parts. The verification operation is
6168 active until the application calls **C_VerifySignature** or **C_VerifySignatureFinal**. To process additional
6169 data (in single or multiple parts), the application MUST call **C_VerifySignatureInit** again.

6170 Any mechanism that supports **C_VerifyInit** must also support **C_VerifySignatureInit**.

6171 **C_VerifySignatureInit** can be called with *pMechanism* set to NULL_PTR to terminate an active
6172 verification operation. If an active operation has been initialized and it cannot be cancelled,
6173 CKR_OPERATION_CANCEL_FAILED must be returned.

6174 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6175 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6176 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6177 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
6178 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
6179 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED,
6180 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
6181 CKR_OPERATION_CANCEL_FAILED.

6182 Example: see **C_VerifySignatureFinal**.

6183 5.15.8 C_VerifySignature

```
6184 CK_DECLARE_FUNCTION(CK_RV, C_VerifySignature) (  
6185     CK_SESSION_HANDLE hSession,  
6186     CK_BYTE_PTR pData,  
6187     CK_ULONG ulDataLen  
6188 );
```

6189 **C_VerifySignature** verifies a signature in a single-part operation, where the signature is an appendix to
6190 the data. *hSession* is the session's handle; *pData* points to the data; *ulDataLen* is the length of the data.

6191 The verification operation MUST have been initialized with **C_VerifySignatureInit**. A call to
6192 **C_VerifySignature** always terminates the active verification operation.

6193 A successful call to **C_VerifySignature** should return either the value CKR_OK (indicating that the
6194 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
6195 invalid). In any of these cases, the active signing operation is terminated.

6196 **C_VerifySignature** cannot be used to terminate a multi-part operation, and MUST be called after
6197 **C_VerifySignatureInit** without intervening **C_VerifySignatureUpdate** calls.

6198 For most mechanisms, **C_VerifySignature** is equivalent to a sequence of **C_VerifySignatureUpdate**
6199 operations followed by **C_VerifySignatureFinal**.

6200 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
6201 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6202 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6203 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
6204 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
6205 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

6206 Example: see **C_VerifySignatureFinal**.

6207 5.15.9 C_VerifySignatureUpdate

```
6208 CK_DECLARE_FUNCTION(CK_RV, C_VerifySignatureUpdate) (  
6209     CK_SESSION_HANDLE hSession,  
6210     CK_BYTE_PTR pPart,  
6211     CK_ULONG ulPartLen  
6212 );
```

6213 **C_VerifySignatureUpdate** continues a multiple-part verification operation, processing another data part.
6214 *hSession* is the session's handle, *pPart* points to the data part; *ulPartLen* is the length of the data part.

6215 The verification operation MUST have been initialized with **C_VerifySignatureInit**. This function may be
6216 called any number of times in succession. A call to **C_VerifySignatureUpdate** which results in an error
6217 terminates the current verification operation.

6218 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6219 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6220 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6221 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
6222 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
6223 CKR_TOKEN_RESOURCE_EXCEEDED.

6224 Example: see **C_VerifySignatureFinal**.

6225 5.15.10 C_VerifySignatureFinal

```
6226 CK_DECLARE_FUNCTION(CK_RV, C_VerifySignatureFinal) (  
6227     CK_SESSION_HANDLE hSession  
6228 );
```

6229 **C_VerifySignatureFinal** finishes a multiple-part verification operation, checking the signature. *hSession*
6230 is the session's handle.

6231 The verification operation MUST have been initialized with **C_VerifySignatureInit**. A call to
6232 **C_VerifySignatureFinal** always terminates the active verification operation.

6233 A successful call to **C_VerifySignatureFinal** should return either the value CKR_OK (indicating that the
6234 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
6235 invalid). In any of these cases, the active verifying operation is terminated.

6236 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6237 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
6238 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6239 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_NOT_INITIALIZED,
6240 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
6241 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

6242 Example:

```
6243 CK_SESSION_HANDLE hSession;
6244 CK_OBJECT_HANDLE hKey;
6245 CK_MECHANISM mechanism = {
6246     CKM_DES_MAC, NULL_PTR, 0
6247 };
6248 CK_BYTE data[] = {...};
6249 CK_BYTE mac[4];
6250 CK_RV rv;
6251 .
6252 .
6253 rv = C_VerifySignatureInit(hSession, &mechanism, hKey, mac, sizeof(mac));
6254 if (rv == CKR_OK) {
6255     rv = C_VerifySignatureUpdate(hSession, data, sizeof(data));
6256     .
6257     .
6258     rv = C_VerifySignatureFinal(hSession);
6259     .
6260     .
6261 }
```

6262 5.16 Message-based functions for verifying signatures and MACs

6263 Message-based verification refers to the process of verifying signatures on multiple messages using the
6264 same verification mechanism and verification key.

6265 Cryptoki provides the following functions for verifying signatures on messages (for the purposes of
6266 Cryptoki, these operations also encompass message authentication codes).

6267 5.16.1 C_MessageVerifyInit

```
6268 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyInit)(
6269     CK_SESSION_HANDLE hSession,
6270     CK_MECHANISM_PTR pMechanism,
6271     CK_OBJECT_HANDLE hKey
6272 );
```

6273 **C_MessageVerifyInit** initializes a message-based verification process, preparing a session for one or
6274 more verification operations (where the signature is an appendix to the data) that use the same
6275 verification mechanism and verification key. *hSession* is the session's handle; *pMechanism* points to the
6276 structure that specifies the verification mechanism; *hKey* is the handle of the verification key.

6277 The **CKA_VERIFY** attribute of the verification key, which indicates whether the key supports verification
6278 where the signature is an appendix to the data, MUST be CK_TRUE.

6279 After calling **C_MessageVerifyInit**, the application can either call **C_VerifyMessage** to verify a signature
6280 on a message in a single part; or call **C_VerifyMessageBegin**, followed by **C_VerifyMessageNext** one
6281 or more times, to verify a signature on a message in multiple parts. This may be repeated several times.

6282 The message-based verification process is active until the application calls **C_MessageVerifyFinal** to
6283 finish the message-based verification process.

6284 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 6285 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
 6286 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 6287 CKR_HOST_MEMORY, CKR_KEY_FUNCTION_NOT_PERMITTED, CKR_KEY_HANDLE_INVALID,
 6288 CKR_KEY_SIZE_RANGE, CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
 6289 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
 6290 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 6291 CKR_USER_NOT_LOGGED_IN.

6292 5.16.2 C_VerifyMessage

```
6293 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessage) (
6294     CK_SESSION_HANDLE hSession,
6295     CK_VOID_PTR pParameter,
6296     CK_ULONG ulParameterLen,
6297     CK_BYTE_PTR pData,
6298     CK_ULONG ulDataLen,
6299     CK_BYTE_PTR pSignature,
6300     CK_ULONG ulSignatureLen
6301 );
```

6302 **C_VerifyMessage** verifies a signature on a message in a single part operation, where the signature is an
 6303 appendix to the data. **C_MessageVerifyInit** must previously been called on the session. *hSession* is the
 6304 session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
 6305 message verification operation; *pData* points to the data; *ulDataLen* is the length of the data; *pSignature*
 6306 points to the signature; *ulSignatureLen* is the length of the signature.

6307 Unlike the *pParameter* parameter of **C_SignMessage**, *pParameter* is always an input parameter.

6308 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**. A call to
 6309 **C_VerifyMessage** starts and terminates a message verification operation.

6310 A successful call to **C_VerifyMessage** should return either the value CKR_OK (indicating that the
 6311 supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that the supplied signature is
 6312 invalid). If the signature can be seen to be invalid purely on the basis of its length, then
 6313 CKR_SIGNATURE_LEN_RANGE should be returned.

6314 **C_VerifyMessage** does not finish the message-based verification process. Additional **C_VerifyMessage**
 6315 or **C_VerifyMessageBegin** and **C_VerifyMessageNext** calls may be made on the session.

6316 For most mechanisms, **C_VerifyMessage** is equivalent to **C_VerifyMessageBegin** followed by a
 6317 sequence of **C_VerifyMessageNext** operations.

6318 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_INVALID,
 6319 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 6320 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 6321 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
 6322 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
 6323 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
 6324 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

6325 5.16.3 C_VerifyMessageBegin

```
6326 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageBegin) (
6327     CK_SESSION_HANDLE hSession,
6328     CK_VOID_PTR pParameter,
6329     CK_ULONG ulParameterLen
```

6330);

6331 **C_VerifyMessageBegin** begins a multiple-part message verification operation, where the signature is an
6332 appendix to the message. **C_MessageVerifyInit** must previously been called on the session. *hSession* is
6333 the session's handle; *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for
6334 the message verification operation.

6335 Unlike the *pParameter* parameter of **C_SignMessageBegin**, *pParameter* is always an input parameter.

6336 After calling **C_VerifyMessageBegin**, the application should call **C_VerifyMessageNext** one or more
6337 times to verify a signature on a message in multiple parts. The message verification operation is active
6338 until the application calls **C_VerifyMessageNext** with a non-NULL *pSignature*. To process additional
6339 messages (in single or multiple parts), the application MUST call **C_VerifyMessage** or
6340 **C_VerifyMessageBegin** again.

6341 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
6342 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
6343 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
6344 CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE,
6345 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_PIN_EXPIRED,
6346 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

6347 5.16.4 C_VerifyMessageNext

```
6348 CK_DECLARE_FUNCTION(CK_RV, C_VerifyMessageNext) (  
6349     CK_SESSION_HANDLE hSession,  
6350     CK_VOID_PTR pParameter,  
6351     CK_ULONG ulParameterLen,  
6352     CK_BYTE_PTR pDataPart,  
6353     CK_ULONG ulDataPartLen,  
6354     CK_BYTE_PTR pSignature,  
6355     CK_ULONG ulSignatureLen  
6356 );
```

6357 **C_VerifyMessageNext** continues a multiple-part message verification operation, processing another data
6358 part, or finishes a multiple-part message verification operation, checking the signature. *hSession* is the
6359 session's handle, *pParameter* and *ulParameterLen* specify any mechanism-specific parameters for the
6360 message verification operation, *pPart* points to the data part; *ulPartLen* is the length of the data part;
6361 *pSignature* points to the signature; *ulSignatureLen* is the length of the signature.

6362 The *pSignature* argument is set to NULL if there is more data part to follow, or set to a non-NULL value
6363 (pointing to the signature to verify) if this is the last data part.

6364 The message verification operation MUST have been started with **C_VerifyMessageBegin**. This function
6365 may be called any number of times in succession. A call to **C_VerifyMessageNext** with a NULL
6366 *pSignature* which results in an error terminates the current message verification operation. A call to
6367 **C_VerifyMessageNext** with a non-NULL *pSignature* always terminates the active message verification
6368 operation.

6369 A successful call to **C_VerifyMessageNext** with a non-NULL *pSignature* should return either the value
6370 CKR_OK (indicating that the supplied signature is valid) or CKR_SIGNATURE_INVALID (indicating that
6371 the supplied signature is invalid). If the signature can be seen to be invalid purely on the basis of its
6372 length, then CKR_SIGNATURE_LEN_RANGE should be returned. In any of these cases, the active
6373 message verifying operation is terminated.

6374 Although the last **C_VerifyMessageNext** call ends the verification of a message, it does not finish the
6375 message-based verification process. Additional **C_VerifyMessage** or **C_VerifyMessageBegin** and
6376 **C_VerifyMessageNext** calls may be made on the session.

6377 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 6378 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 6379 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 6380 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
 6381 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
 6382 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SIGNATURE_INVALID,
 6383 CKR_SIGNATURE_LEN_RANGE, CKR_TOKEN_RESOURCE_EXCEEDED.

6384 5.16.5 C_MessageVerifyFinal

```
6385 CK_DECLARE_FUNCTION(CK_RV, C_MessageVerifyFinal) (
6386     CK_SESSION_HANDLE hSession
6387 );
```

6388 **C_MessageVerifyFinal** finishes a message-based verification process. *hSession* is the session's handle.

6389 The message-based verification process MUST have been initialized with **C_MessageVerifyInit**.

6390 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
 6391 CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
 6392 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
 6393 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
 6394 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
 6395 CKR_SESSION_HANDLE_INVALID, CKR_TOKEN_RESOURCE_EXCEEDED.

6396 5.17 Dual-function cryptographic functions

6397 Cryptoki provides the following functions to perform two cryptographic operations “simultaneously” within
 6398 a session. These functions are provided so as to avoid unnecessarily passing data back and forth to and
 6399 from a token.

6400 5.17.1 C_DigestEncryptUpdate

```
6401 CK_DECLARE_FUNCTION(CK_RV, C_DigestEncryptUpdate) (
6402     CK_SESSION_HANDLE hSession,
6403     CK_BYTE_PTR pPart,
6404     CK_ULONG ulPartLen,
6405     CK_BYTE_PTR pEncryptedPart,
6406     CK_ULONG_PTR pulEncryptedPartLen
6407 );
```

6408 **C_DigestEncryptUpdate** continues multiple-part digest and encryption operations, processing another
 6409 data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the
 6410 data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part;
 6411 *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

6412 **C_DigestEncryptUpdate** uses the convention described in Section 5.2 on producing output. If a
 6413 **C_DigestEncryptUpdate** call does not produce encrypted output (because an error occurs, or because
 6414 *pEncryptedPart* has the value NULL_PTR, or because *pulEncryptedPartLen* is too small to hold the entire
 6415 encrypted part output), then no plaintext is passed to the active digest operation.

6416 Digest and encryption operations MUST both be active (they MUST have been initialized with
 6417 **C_DigestInit** and **C_EncryptInit**, respectively). This function may be called any number of times in
 6418 succession, and may be interspersed with **C_DigestUpdate**, **C_DigestKey**, and **C_EncryptUpdate** calls
 6419 (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to
 6420 **C_DigestKey**, however).

6421 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
 6422 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
 6423 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
 6424 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,

6425 CKR_OPERATION_ACTIVE, CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING,
6426 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID.

6427 Example:

```
6428 #define BUF_SZ 512
6429
6430 CK_SESSION_HANDLE hSession;
6431 CK_OBJECT_HANDLE hKey;
6432 CK_BYTE iv[8];
6433 CK_MECHANISM digestMechanism = {
6434     CKM_MD5, NULL_PTR, 0
6435 };
6436 CK_MECHANISM encryptionMechanism = {
6437     CKM_DES_ECB, iv, sizeof(iv)
6438 };
6439 CK_BYTE encryptedData[BUF_SZ];
6440 CK_ULONG ulEncryptedDataLen;
6441 CK_BYTE digest[16];
6442 CK_ULONG ulDigestLen;
6443 CK_BYTE data[(2*BUF_SZ)+8];
6444 CK_RV rv;
6445 int i;
6446
6447 .
6448 .
6449 memset(iv, 0, sizeof(iv));
6450 memset(data, 'A', ((2*BUF_SZ)+5));
6451 rv = C_EncryptInit(hSession, &encryptionMechanism, hKey);
6452 if (rv != CKR_OK) {
6453     .
6454     .
6455 }
6456 rv = C_DigestInit(hSession, &digestMechanism);
6457 if (rv != CKR_OK) {
6458     .
6459     .
6460 }
6461
6462 ulEncryptedDataLen = sizeof(encryptedData);
6463 rv = C_DigestEncryptUpdate(
6464     hSession,
6465     &data[0], BUF_SZ,
6466     encryptedData, &ulEncryptedDataLen);
6467 .
```

```

6468 .
6469 ulEncryptedDataLen = sizeof(encryptedData);
6470 rv = C_DigestEncryptUpdate(
6471     hSession,
6472     &data[BUF_SZ], BUF_SZ,
6473     encryptedData, &ulEncryptedDataLen);
6474 .
6475 .
6476
6477 /*
6478  * The last portion of the buffer needs to be
6479  * handled with separate calls to deal with
6480  * padding issues in ECB mode
6481  */
6482
6483 /* First, complete the digest on the buffer */
6484 rv = C_DigestUpdate(hSession, &data[BUF_SZ*2], 5);
6485 .
6486 .
6487 ulDigestLen = sizeof(digest);
6488 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
6489 .
6490 .
6491
6492 /* Then, pad last part with 3 0x00 bytes, and complete encryption */
6493 for(i=0;i<3;i++)
6494     data[((BUF_SZ*2)+5)+i] = 0x00;
6495
6496 /* Now, get second-to-last piece of ciphertext */
6497 ulEncryptedDataLen = sizeof(encryptedData);
6498 rv = C_EncryptUpdate(
6499     hSession,
6500     &data[BUF_SZ*2], 8,
6501     encryptedData, &ulEncryptedDataLen);
6502 .
6503 .
6504
6505 /* Get last piece of ciphertext (should have length 0, here) */
6506 ulEncryptedDataLen = sizeof(encryptedData);
6507 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
6508 .
6509 .

```

5.17.2 C_DecryptDigestUpdate

```
CK_DECLARE_FUNCTION(CK_RV, C_DecryptDigestUpdate) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG ulEncryptedPartLen,
    CK_BYTE_PTR pPart,
    CK_ULONG_PTR pulPartLen
);
```

C_DecryptDigestUpdate continues a multiple-part combined decryption and digest operation, processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted data part; *ulEncryptedPartLen* is the length of the encrypted data part; *pPart* points to the location that receives the recovered data part; *pulPartLen* points to the location that holds the length of the recovered data part.

C_DecryptDigestUpdate uses the convention described in Section 5.2 on producing output. If a **C_DecryptDigestUpdate** 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 decrypted part output), then no plaintext is passed to the active digest operation.

Decryption and digesting operations **MUST** both be active (they **MUST** have been initialized with **C_DecryptInit** and **C_DigestInit**, respectively). This function may be called any number of times in succession, and may be interspersed with **C_DecryptUpdate**, **C_DigestUpdate**, and **C_DigestKey** calls (it would be somewhat unusual to intersperse calls to **C_DigestEncryptUpdate** with calls to **C_DigestKey**, however).

Use of **C_DecryptDigestUpdate** involves a pipelining issue that does not arise when using **C_DigestEncryptUpdate**, the "inverse function" of **C_DecryptDigestUpdate**. This is because when **C_DigestEncryptUpdate** is called, precisely the same input is passed to both the active digesting operation and the active encryption operation; however, when **C_DecryptDigestUpdate** is called, the input passed to the active digesting 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 digest the original plaintext thereby obtained.

After initializing decryption and digesting operations, the application passes the 24-byte ciphertext (3 DES blocks) into **C_DecryptDigestUpdate**. **C_DecryptDigestUpdate** 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 digesting 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 digesting operations are linked *only* through the **C_DecryptDigestUpdate** call, these 2 bytes of plaintext are *not* passed on to be digested.

A call to **C_DigestFinal**, therefore, would compute the message digest of *the first 16 bytes of the plaintext*, not the message digest of the entire plaintext. It is crucial that, before **C_DigestFinal** is called, the last 2 bytes of plaintext get passed into the active digesting operation via a **C_DigestUpdate** call.

Because of this, it is critical that when an application uses a padded decryption mechanism with **C_DecryptDigestUpdate**, it knows exactly how much plaintext has been passed into the active digesting operation. *Extreme caution is warranted when using a padded decryption mechanism with C_DecryptDigestUpdate.*

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_ACTIVE`,

6561 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
6562 CKR_SESSION_HANDLE_INVALID.

6563 Example:

```
6564 #define BUF_SZ 512
6565
6566 CK_SESSION_HANDLE hSession;
6567 CK_OBJECT_HANDLE hKey;
6568 CK_BYTE iv[8];
6569 CK_MECHANISM decryptionMechanism = {
6570     CKM_DES_ECB, iv, sizeof(iv)
6571 };
6572 CK_MECHANISM digestMechanism = {
6573     CKM_MD5, NULL_PTR, 0
6574 };
6575 CK_BYTE encryptedData[(2*BUF_SZ)+8];
6576 CK_BYTE digest[16];
6577 CK_ULONG ulDigestLen;
6578 CK_BYTE data[BUF_SZ];
6579 CK_ULONG ulDataLen, ulLastUpdateSize;
6580 CK_RV rv;
6581
6582 .
6583 .
6584 memset(iv, 0, sizeof(iv));
6585 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
6586 rv = C_DecryptInit(hSession, &decryptionMechanism, hKey);
6587 if (rv != CKR_OK) {
6588     .
6589     .
6590 }
6591 rv = C_DigestInit(hSession, &digestMechanism);
6592 if (rv != CKR_OK){
6593     .
6594     .
6595 }
6596
6597 ulDataLen = sizeof(data);
6598 rv = C_DecryptDigestUpdate(
6599     hSession,
6600     &encryptedData[0], BUF_SZ,
6601     data, &ulDataLen);
6602 .
6603 .
```

```

6604 ulDataLen = sizeof(data);
6605 rv = C_DecryptDigestUpdate(
6606     hSession,
6607     &encryptedData[BUF_SZ], BUF_SZ,
6608     data, &ulDataLen);
6609 .
6610 .
6611
6612 /*
6613  * The last portion of the buffer needs to be handled with
6614  * separate calls to deal with padding issues in ECB mode
6615  */
6616
6617 /* First, complete the decryption of the buffer */
6618 ulLastUpdateSize = sizeof(data);
6619 rv = C_DecryptUpdate(
6620     hSession,
6621     &encryptedData[BUF_SZ*2], 8,
6622     data, &ulLastUpdateSize);
6623 .
6624 .
6625 /* Get last piece of plaintext (should have length 0, here) */
6626 ulDataLen = sizeof(data)-ulLastUpdateSize;
6627 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
6628 if (rv != CKR_OK) {
6629     .
6630     .
6631 }
6632
6633 /* Digest last bit of plaintext */
6634 rv = C_DigestUpdate(hSession, data, 5);
6635 if (rv != CKR_OK) {
6636     .
6637     .
6638 }
6639 ulDigestLen = sizeof(digest);
6640 rv = C_DigestFinal(hSession, digest, &ulDigestLen);
6641 if (rv != CKR_OK) {
6642     .
6643     .
6644 }

```

5.17.3 C_SignEncryptUpdate

```
CK_DECLARE_FUNCTION(CK_RV, C_SignEncryptUpdate) (
    CK_SESSION_HANDLE hSession,
    CK_BYTE_PTR pPart,
    CK_ULONG ulPartLen,
    CK_BYTE_PTR pEncryptedPart,
    CK_ULONG_PTR pulEncryptedPartLen
);
```

C_SignEncryptUpdate continues a multiple-part combined signature and encryption operation, processing another data part. *hSession* is the session's handle; *pPart* points to the data part; *ulPartLen* is the length of the data part; *pEncryptedPart* points to the location that receives the digested and encrypted data part; and *pulEncryptedPartLen* points to the location that holds the length of the encrypted data part.

C_SignEncryptUpdate uses the convention described in Section 5.2 on producing output. If a **C_SignEncryptUpdate** call does not produce encrypted output (because an error occurs, or because *pEncryptedPart* has the value `NULL_PTR`, or because *pulEncryptedPartLen* is too small to hold the entire encrypted part output), then no plaintext is passed to the active signing operation.

Signature and encryption operations **MUST** both be active (they **MUST** have been initialized with **C_SignInit** and **C_EncryptInit**, respectively). This function may be called any number of times in succession, and may be interspersed with **C_SignUpdate** and **C_EncryptUpdate** calls.

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_FUNCTION_CANCELED`, `CKR_FUNCTION_FAILED`, `CKR_GENERAL_ERROR`, `CKR_HOST_MEMORY`, `CKR_OK`, `CKR_OPERATION_ACTIVE`, `CKR_OPERATION_NOT_INITIALIZED`, `CKR_PENDING`, `CKR_SESSION_CLOSED`, `CKR_SESSION_HANDLE_INVALID`, `CKR_USER_NOT_LOGGED_IN`.

Example:

```
#define BUF_SZ 512

CK_SESSION_HANDLE hSession;
CK_OBJECT_HANDLE hEncryptionKey, hMacKey;
CK_BYTE iv[8];
CK_MECHANISM signMechanism = {
    CKM_DES_MAC, NULL_PTR, 0
};
CK_MECHANISM encryptionMechanism = {
    CKM_DES_ECB, iv, sizeof(iv)
};
CK_BYTE encryptedData[BUF_SZ];
CK_ULONG ulEncryptedDataLen;
CK_BYTE MAC[4];
CK_ULONG ulMacLen;
CK_BYTE data[(2*BUF_SZ)+8];
CK_RV rv;
int i;

.
```

```

6692 memset(iv, 0, sizeof(iv));
6693 memset(data, 'A', ((2*BUF_SZ)+5));
6694 rv = C_EncryptInit(hSession, &encryptionMechanism, hEncryptionKey);
6695 if (rv != CKR_OK) {
6696     .
6697     .
6698 }
6699 rv = C_SignInit(hSession, &signMechanism, hMacKey);
6700 if (rv != CKR_OK) {
6701     .
6702     .
6703 }
6704
6705 ulEncryptedDataLen = sizeof(encryptedData);
6706 rv = C_SignEncryptUpdate(
6707     hSession,
6708     &data[0], BUF_SZ,
6709     encryptedData, &ulEncryptedDataLen);
6710 .
6711 .
6712 ulEncryptedDataLen = sizeof(encryptedData);
6713 rv = C_SignEncryptUpdate(
6714     hSession,
6715     &data[BUF_SZ], BUF_SZ,
6716     encryptedData, &ulEncryptedDataLen);
6717 .
6718 .
6719
6720 /*
6721  * The last portion of the buffer needs to be handled with
6722  * separate calls to deal with padding issues in ECB mode
6723  */
6724
6725 /* First, complete the signature on the buffer */
6726 rv = C_SignUpdate(hSession, &data[BUF_SZ*2], 5);
6727 .
6728 .
6729 ulMacLen = sizeof(MAC);
6730 rv = C_SignFinal(hSession, MAC, &ulMacLen);
6731 .
6732 .
6733
6734 /* Then pad last part with 3 0x00 bytes, and complete encryption */

```

```

6735 for(i=0;i<3;i++)
6736     data[((BUF_SZ*2)+5)+i] = 0x00;
6737
6738 /* Now, get second-to-last piece of ciphertext */
6739 ulEncryptedDataLen = sizeof(encryptedData);
6740 rv = C_EncryptUpdate(
6741     hSession,
6742     &data[BUF_SZ*2], 8,
6743     encryptedData, &ulEncryptedDataLen);
6744 .
6745 .
6746
6747 /* Get last piece of ciphertext (should have length 0, here) */
6748 ulEncryptedDataLen = sizeof(encryptedData);
6749 rv = C_EncryptFinal(hSession, encryptedData, &ulEncryptedDataLen);
6750 .
6751 .

```

6752 5.17.4 C_DecryptVerifyUpdate

```

6753 CK_DECLARE_FUNCTION(CK_RV, C_DecryptVerifyUpdate) (
6754     CK_SESSION_HANDLE hSession,
6755     CK_BYTE_PTR pEncryptedPart,
6756     CK_ULONG ulEncryptedPartLen,
6757     CK_BYTE_PTR pPart,
6758     CK_ULONG_PTR pulPartLen
6759 );

```

6760 **C_DecryptVerifyUpdate** continues a multiple-part combined decryption and verification operation,
6761 processing another data part. *hSession* is the session's handle; *pEncryptedPart* points to the encrypted
6762 data; *ulEncryptedPartLen* is the length of the encrypted data; *pPart* points to the location that receives the
6763 recovered data; and *pulPartLen* points to the location that holds the length of the recovered data.

6764 **C_DecryptVerifyUpdate** uses the convention described in Section 5.2 on producing output. If a
6765 **C_DecryptVerifyUpdate** call does not produce decrypted output (because an error occurs, or because
6766 *pPart* has the value `NULL_PTR`, or because *pulPartLen* is too small to hold the entire encrypted part
6767 output), then no plaintext is passed to the active verification operation.

6768 Decryption and signature operations **MUST** both be active (they **MUST** have been initialized with
6769 **C_DecryptInit** and **C_VerifyInit**, respectively). This function may be called any number of times in
6770 succession, and may be interspersed with **C_DecryptUpdate** and **C_VerifyUpdate** calls.

6771 Use of **C_DecryptVerifyUpdate** involves a pipelining issue that does not arise when using
6772 **C_SignEncryptUpdate**, the "inverse function" of **C_DecryptVerifyUpdate**. This is because when
6773 **C_SignEncryptUpdate** is called, precisely the same input is passed to both the active signing operation
6774 and the active encryption operation; however, when **C_DecryptVerifyUpdate** is called, the input passed
6775 to the active verifying operation is the *output* of the active decryption operation. This issue comes up only
6776 when the mechanism used for decryption performs padding.

6777 In particular, envision a 24-byte ciphertext which was obtained by encrypting an 18-byte plaintext with
6778 DES in CBC mode with PKCS padding. Consider an application which will simultaneously decrypt this
6779 ciphertext and verify a signature on the original plaintext thereby obtained.

6780 After initializing decryption and verification operations, the application passes the 24-byte ciphertext (3
6781 DES blocks) into **C_DecryptVerifyUpdate**. **C_DecryptVerifyUpdate** returns exactly 16 bytes of plaintext,

6782 since at this point, Cryptoki doesn't know if there's more ciphertext coming, or if the last block of
6783 ciphertext held any padding. These 16 bytes of plaintext are passed into the active verification operation.

6784 Since there is no more ciphertext, the application calls **C_DecryptFinal**. This tells Cryptoki that there's no
6785 more ciphertext coming, and the call returns the last 2 bytes of plaintext. However, since the active
6786 decryption and verification operations are linked *only* through the **C_DecryptVerifyUpdate** call, these 2
6787 bytes of plaintext are *not* passed on to the verification mechanism.

6788 A call to **C_VerifyFinal**, therefore, would verify whether or not the signature supplied is a valid signature
6789 on *the first 16 bytes of the plaintext*, not on the entire plaintext. It is crucial that, before **C_VerifyFinal** is
6790 called, the last 2 bytes of plaintext get passed into the active verification operation via a **C_VerifyUpdate**
6791 call.

6792 Because of this, it is critical that when an application uses a padded decryption mechanism with
6793 **C_DecryptVerifyUpdate**, it knows exactly how much plaintext has been passed into the active
6794 verification operation. *Extreme caution is warranted when using a padded decryption mechanism with*
6795 **C_DecryptVerifyUpdate**.

6796 Return values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
6797 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DATA_LEN_RANGE, CKR_DEVICE_ERROR,
6798 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_ENCRYPTED_DATA_INVALID,
6799 CKR_ENCRYPTED_DATA_LEN_RANGE, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
6800 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE,
6801 CKR_OPERATION_NOT_INITIALIZED, CKR_PENDING, CKR_SESSION_CLOSED,
6802 CKR_SESSION_HANDLE_INVALID.

6803 Example:

```

6804 #define BUF_SZ 512
6805
6806 CK_SESSION_HANDLE hSession;
6807 CK_OBJECT_HANDLE hDecryptionKey, hMacKey;
6808 CK_BYTE iv[8];
6809 CK_MECHANISM decryptionMechanism = {
6810     CKM_DES_ECB, iv, sizeof(iv)
6811 };
6812 CK_MECHANISM verifyMechanism = {
6813     CKM_DES_MAC, NULL_PTR, 0
6814 };
6815 CK_BYTE encryptedData[(2*BUF_SZ)+8];
6816 CK_BYTE MAC[4];
6817 CK_ULONG ulMacLen;
6818 CK_BYTE data[BUF_SZ];
6819 CK_ULONG ulDataLen, ulLastUpdateSize;
6820 CK_RV rv;
6821
6822 .
6823 .
6824 memset(iv, 0, sizeof(iv));
6825 memset(encryptedData, 'A', ((2*BUF_SZ)+8));
6826 rv = C_DecryptInit(hSession, &decryptionMechanism, hDecryptionKey);
6827 if (rv != CKR_OK) {
6828     .

```

```

6829     .
6830 }
6831 rv = C_VerifyInit(hSession, &verifyMechanism, hMacKey);
6832 if (rv != CKR_OK) {
6833     .
6834     .
6835 }
6836
6837 ulDataLen = sizeof(data);
6838 rv = C_DecryptVerifyUpdate(
6839     hSession,
6840     &encryptedData[0], BUF_SZ,
6841     data, &ulDataLen);
6842 .
6843 .
6844 ulDataLen = sizeof(data);
6845 rv = C_DecryptVerifyUpdate(
6846     hSession,
6847     &encryptedData[BUF_SZ], BUF_SZ,
6848     data, &ulDataLen);
6849 .
6850 .
6851
6852 /*
6853  * The last portion of the buffer needs to be handled with
6854  * separate calls to deal with padding issues in ECB mode
6855  */
6856
6857 /* First, complete the decryption of the buffer */
6858 ulLastUpdateSize = sizeof(data);
6859 rv = C_DecryptUpdate(
6860     hSession,
6861     &encryptedData[BUF_SZ*2], 8,
6862     data, &ulLastUpdateSize);
6863 .
6864 .
6865 /* Get last little piece of plaintext. Should have length 0 */
6866 ulDataLen = sizeof(data)-ulLastUpdateSize;
6867 rv = C_DecryptFinal(hSession, &data[ulLastUpdateSize], &ulDataLen);
6868 if (rv != CKR_OK) {
6869     .
6870     .
6871 }

```

```

6872
6873 /* Send last bit of plaintext to verification operation */
6874 rv = C_VerifyUpdate(hSession, data, 5);
6875 if (rv != CKR_OK) {
6876     .
6877     .
6878 }
6879 rv = C_VerifyFinal(hSession, MAC, ulMacLen);
6880 if (rv == CKR_SIGNATURE_INVALID) {
6881     .
6882     .
6883 }

```

6884 5.18 Key management functions

6885 Cryptoki provides the following functions for key management:

6886 5.18.1 C_GenerateKey

```

6887 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKey) (
6888     CK_SESSION_HANDLE hSession
6889     CK_MECHANISM_PTR pMechanism,
6890     CK_ATTRIBUTE_PTR pTemplate,
6891     CK_ULONG ulCount,
6892     CK_OBJECT_HANDLE_PTR phKey
6893 );

```

6894 **C_GenerateKey** generates a secret key or set of domain parameters, creating a new object. *hSession* is
6895 the session's handle; *pMechanism* points to the generation mechanism; *pTemplate* points to the template
6896 for the new key or set of domain parameters; *ulCount* is the number of attributes in the template; *phKey*
6897 points to the location that receives the handle of the new key or set of domain parameters.

6898 If the generation mechanism is for domain parameter generation, the **CKA_CLASS** attribute will have the
6899 value **CKO_DOMAIN_PARAMETERS**; otherwise, it will have the value **CKO_SECRET_KEY**.

6900 Since the type of key or domain parameters to be generated is implicit in the generation mechanism, the
6901 template does not need to supply a key type. If it does supply a key type which is inconsistent with the
6902 generation mechanism, **C_GenerateKey** fails and returns the error code
6903 **CKR_TEMPLATE_INCONSISTENT**. The **CKA_CLASS** attribute is treated similarly.

6904 If a call to **C_GenerateKey** cannot support the precise template supplied to it, it will fail and return without
6905 creating an object.

6906 The object created by a successful call to **C_GenerateKey** will have its **CKA_LOCAL** attribute set to
6907 CK_TRUE. In addition, the object created will have a value for **CKA_UNIQUE_ID** generated and
6908 assigned (See Section 4.4.1).

6909 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
6910 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
6911 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
6912 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED,
6913 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
6914 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
6915 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
6916 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
6917 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
6918 CKR_USER_NOT_LOGGED_IN.

6919 Example:

```
6920 CK_SESSION_HANDLE hSession;
6921 CK_OBJECT_HANDLE hKey;
6922 CK_MECHANISM mechanism = {
6923     CKM_DES_KEY_GEN, NULL_PTR, 0
6924 };
6925 CK_RV rv;
6926
6927 .
6928 .
6929 rv = C_GenerateKey(hSession, &mechanism, NULL_PTR, 0, &hKey);
6930 if (rv == CKR_OK) {
6931     .
6932     .
6933 }
```

6934 5.18.2 C_GenerateKeyPair

```
6935 CK_DECLARE_FUNCTION(CK_RV, C_GenerateKeyPair)(
6936     CK_SESSION_HANDLE hSession,
6937     CK_MECHANISM_PTR pMechanism,
6938     CK_ATTRIBUTE_PTR pPublicKeyTemplate,
6939     CK_ULONG ulPublicKeyAttributeCount,
6940     CK_ATTRIBUTE_PTR pPrivateKeyTemplate,
6941     CK_ULONG ulPrivateKeyAttributeCount,
6942     CK_OBJECT_HANDLE_PTR phPublicKey,
6943     CK_OBJECT_HANDLE_PTR phPrivateKey
6944 );
```

6945 **C_GenerateKeyPair** generates a public/private key pair, creating new key objects. *hSession* is the
6946 session's handle; *pMechanism* points to the key generation mechanism; *pPublicKeyTemplate* points to
6947 the template for the public key; *ulPublicKeyAttributeCount* is the number of attributes in the public-key
6948 template; *pPrivateKeyTemplate* points to the template for the private key; *ulPrivateKeyAttributeCount* is
6949 the number of attributes in the private-key template; *phPublicKey* points to the location that receives the
6950 handle of the new public key; *phPrivateKey* points to the location that receives the handle of the new
6951 private key.

6952 Since the types of keys to be generated are implicit in the key pair generation mechanism, the templates
6953 do not need to supply key types. If one of the templates does supply a key type which is inconsistent with
6954 the key generation mechanism, **C_GenerateKeyPair** fails and returns the error code
6955 **CKR_TEMPLATE_INCONSISTENT**. The **CKA_CLASS** attribute is treated similarly.

6956 If a call to **C_GenerateKeyPair** cannot support the precise templates supplied to it, it will fail and return
6957 without creating any key objects.

6958 A call to **C_GenerateKeyPair** will never create just one key and return. A call can fail, and create no
6959 keys; or it can succeed, and create a matching public/private key pair.

6960 The key objects created by a successful call to **C_GenerateKeyPair** will have their **CKA_LOCAL**
6961 attributes set to CK_TRUE. In addition, the key objects created will both have values for
6962 **CKA_UNIQUE_ID** generated and assigned (See Section 4.4.1).

6963 *Note carefully the order of the arguments to C_GenerateKeyPair. The last two arguments do not have*
6964 *the same order as they did in the original Cryptoki Version 1.0 document. The order of these two*
6965 *arguments has caused some unfortunate confusion.*

6966 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
 6967 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
 6968 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
 6969 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
 6970 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 6971 CKR_HOST_MEMORY, CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID,
 6972 CKR_OK, CKR_OPERATION_ACTIVE, CKR_PARAMETER_SET_NOT_SUPPORTED,
 6973 CKR_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
 6974 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
 6975 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

6976 Example:

```

6977 CK_SESSION_HANDLE hSession;
6978 CK_OBJECT_HANDLE hPublicKey, hPrivateKey;
6979 CK_MECHANISM mechanism = {
6980     CKM_RSA_PKCS_KEY_PAIR_GEN, NULL_PTR, 0
6981 };
6982 CK_ULONG modulusBits = 3072;
6983 CK_BYTE publicExponent[] = { 3 };
6984 CK_BYTE subject[] = {...};
6985 CK_BYTE id[] = {123};
6986 CK_BBOOL true = CK_TRUE;
6987 CK_ATTRIBUTE publicKeyTemplate[] = {
6988     {CKA_ENCRYPT, &true, sizeof(true)},
6989     {CKA_VERIFY, &true, sizeof(true)},
6990     {CKA_WRAP, &true, sizeof(true)},
6991     {CKA_MODULUS_BITS, &modulusBits, sizeof(modulusBits)},
6992     {CKA_PUBLIC_EXPONENT, publicExponent, sizeof(publicExponent)}
6993 };
6994 CK_ATTRIBUTE privateKeyTemplate[] = {
6995     {CKA_TOKEN, &true, sizeof(true)},
6996     {CKA_PRIVATE, &true, sizeof(true)},
6997     {CKA_SUBJECT, subject, sizeof(subject)},
6998     {CKA_ID, id, sizeof(id)},
6999     {CKA_SENSITIVE, &true, sizeof(true)},
7000     {CKA_DECRYPT, &true, sizeof(true)},
7001     {CKA_SIGN, &true, sizeof(true)},
7002     {CKA_UNWRAP, &true, sizeof(true)}
7003 };
7004 CK_RV rv;
7005
7006 rv = C_GenerateKeyPair(
7007     hSession, &mechanism,
7008     publicKeyTemplate, 5,
7009     privateKeyTemplate, 8,
7010     &hPublicKey, &hPrivateKey);

```

```

7011 if (rv == CKR_OK) {
7012     .
7013     .
7014 }

```

7015 5.18.3 C_WrapKey

```

7016 CK_DECLARE_FUNCTION(CK_RV, C_WrapKey) (
7017     CK_SESSION_HANDLE hSession,
7018     CK_MECHANISM_PTR pMechanism,
7019     CK_OBJECT_HANDLE hWrappingKey,
7020     CK_OBJECT_HANDLE hKey,
7021     CK_BYTE_PTR pWrappedKey,
7022     CK_ULONG_PTR pulWrappedKeyLen
7023 );

```

7024 **C_WrapKey** wraps (i.e., encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism*
7025 points to the wrapping mechanism; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle
7026 of the key to be wrapped; *pWrappedKey* points to the location that receives the wrapped key; and
7027 *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

7028 **C_WrapKey** uses the convention described in Section 5.2 on producing output.

7029 The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping,
7030 MUST be CK_TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped MUST also be
7031 CK_TRUE.

7032 If the key to be wrapped cannot be wrapped for some token-specific reason, despite it having its
7033 **CKA_EXTRACTABLE** attribute set to CK_TRUE, then **C_WrapKey** fails with error code
7034 **CKR_KEY_NOT_WRAPPABLE**. If it cannot be wrapped with the specified wrapping key and mechanism
7035 solely because of its length, then **C_WrapKey** fails with error code **CKR_KEY_SIZE_RANGE**.

7036 **C_WrapKey** can be used in the following situations:

- 7037 • To wrap any secret key with a public key that supports encryption and decryption.
- 7038 • To wrap any secret key with any other secret key. Consideration MUST be given to key size and
7039 mechanism strength or the token may not allow the operation.
- 7040 • To wrap a private key with any secret key.

7041 Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

7042 To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute
7043 **CKA_WRAP_TEMPLATE** can be used on the wrapping key to specify an attribute set that will be
7044 compared against the attributes of the key to be wrapped. If all attributes match according to the
7045 **C_FindObject** rules of attribute matching then the wrap will proceed. The value of this attribute is an
7046 attribute template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If
7047 this attribute is not supplied then any template is acceptable. If an attribute is not present, it will not be
7048 checked. If any attribute mismatch occurs on an attempt to wrap a key then the function SHALL return
7049 **CKR_KEY_HANDLE_INVALID**.

7050 Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
7051 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
7052 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
7053 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
7054 CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
7055 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
7056 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
7057 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
7058 CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE,
7059 CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

7060 Example:

```
7061 CK_SESSION_HANDLE hSession;
7062 CK_OBJECT_HANDLE hWrappingKey, hKey;
7063 CK_MECHANISM mechanism = {
7064     CKM_DES3_ECB, NULL_PTR, 0
7065 };
7066 CK_BYTE wrappedKey[8];
7067 CK_ULONG ulWrappedKeyLen;
7068 CK_RV rv;
7069
7070 .
7071 .
7072 ulWrappedKeyLen = sizeof(wrappedKey);
7073 rv = C_WrapKey(
7074     hSession, &mechanism,
7075     hWrappingKey, hKey,
7076     wrappedKey, &ulWrappedKeyLen);
7077 if (rv == CKR_OK) {
7078     .
7079     .
7080 }
```

7081 5.18.4 C_UnwrapKey

```
7082 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKey) (
7083     CK_SESSION_HANDLE hSession,
7084     CK_MECHANISM_PTR pMechanism,
7085     CK_OBJECT_HANDLE hUnwrappingKey,
7086     CK_BYTE_PTR pWrappedKey,
7087     CK_ULONG ulWrappedKeyLen,
7088     CK_ATTRIBUTE_PTR pTemplate,
7089     CK_ULONG ulAttributeCount,
7090     CK_OBJECT_HANDLE_PTR phKey
7091 );
```

7092 **C_UnwrapKey** unwraps (i.e. decrypts) a wrapped key, creating a new private key or secret key object.
7093 *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism; *hUnwrappingKey* is
7094 the handle of the unwrapping key; *pWrappedKey* points to the wrapped key; *ulWrappedKeyLen* is the
7095 length of the wrapped key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the
7096 number of attributes in the template; *phKey* points to the location that receives the handle of the
7097 recovered key.

7098 The **CKA_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports
7099 unwrapping, MUST be CK_TRUE.

7100 The template for the new key SHALL specify **CKA_VALUE_LEN** when neither the key type of the
7101 unwrapped key nor the unwrapping mechanism unambiguously determine the length of the unwrapped
7102 key; otherwise, the function SHALL return **CKR_TEMPLATE_INCOMPLETE**.

7103 The template for the new key MAY specify **CKA_VALUE_LEN** when the key type of the unwrapped key
7104 or the unwrapping mechanism unambiguously determine the length of the unwrapped key. If any length
7105 conflict occurs between the key type of the unwrapped key, the output from the unwrapping mechanism,

7106 or the specified **CKA_VALUE_LEN**, then the function SHALL return
7107 **CKR_WRAPPED_KEY_LEN_RANGE**.

7108 The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the
7109 **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by
7110 default set to CK_TRUE.

7111 Some mechanisms may modify, or attempt to modify, the contents of the pMechanism structure at the
7112 same time that the key is unwrapped.

7113 If a call to **C_UnwrapKey** cannot support the precise template supplied to it, it will fail and return without
7114 creating any key object.

7115 The key object created by a successful call to **C_UnwrapKey** will have its **CKA_LOCAL** attribute set to
7116 CK_FALSE. In addition, the object created will have a value for **CKA_UNIQUE_ID** generated and
7117 assigned (See Section 4.4.1).

7118 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute
7119 **CKA_UNWRAP_TEMPLATE** can be used on the unwrapping key to specify an attribute set that will be
7120 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied
7121 attribute template, in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute
7122 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
7123 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute
7124 conflict occurs on an attempt to unwrap a key then the function SHALL return
7125 **CKR_TEMPLATE_INCONSISTENT**.

7126 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
7127 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
7128 CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
7129 CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
7130 CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED,
7131 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
7132 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
7133 CKR_OPERATION_ACTIVE, CKR_PARAMETER_SET_NOT_SUPPORTED, CKR_PENDING,
7134 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
7135 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
7136 CKR_TOKEN_WRITE_PROTECTED, CKR_UNWRAPPING_KEY_HANDLE_INVALID,
7137 CKR_UNWRAPPING_KEY_SIZE_RANGE, CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT,
7138 CKR_USER_NOT_LOGGED_IN, CKR_WRAPPED_KEY_INVALID,
7139 CKR_WRAPPED_KEY_LEN_RANGE.

7140 Example:

```

7141 CK_SESSION_HANDLE hSession;
7142 CK_OBJECT_HANDLE hUnwrappingKey, hKey;
7143 CK_MECHANISM mechanism = {
7144     CKM_DES3_ECB, NULL_PTR, 0
7145 };
7146 CK_BYTE wrappedKey[8] = {...};
7147 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
7148 CK_KEY_TYPE keyType = CKK_DES;
7149 CK_BBOOL true = CK_TRUE;
7150 CK_ATTRIBUTE template[] = {
7151     {CKA_CLASS, &keyClass, sizeof(keyClass)},
7152     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7153     {CKA_ENCRYPT, &true, sizeof(true)},
7154     {CKA_DECRYPT, &true, sizeof(true)}

```

```

7155 };
7156 CK_RV rv;
7157
7158 .
7159 .
7160 rv = C_UnwrapKey(
7161     hSession, &mechanism, hUnwrappingKey,
7162     wrappedKey, sizeof(wrappedKey), template, 4, &hKey);
7163 if (rv == CKR_OK) {
7164     .
7165     .
7166 }

```

7167 5.18.5 C_DeriveKey

```

7168 CK_DECLARE_FUNCTION(CK_RV, C_DeriveKey)(
7169     CK_SESSION_HANDLE hSession,
7170     CK_MECHANISM_PTR pMechanism,
7171     CK_OBJECT_HANDLE hBaseKey,
7172     CK_ATTRIBUTE_PTR pTemplate,
7173     CK_ULONG ulAttributeCount,
7174     CK_OBJECT_HANDLE_PTR phKey
7175 );

```

7176 **C_DeriveKey** derives a key from a base key, creating a new key object. *hSession* is the session's
7177 handle; *pMechanism* points to a structure that specifies the key derivation mechanism; *hBaseKey* is the
7178 handle of the base key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number
7179 of attributes in the template; and *phKey* points to the location that receives the handle of the derived key.

7180 The values of the **CKA_SENSITIVE**, **CKA_ALWAYS_SENSITIVE**, **CKA_EXTRACTABLE**, and
7181 **CKA_NEVER_EXTRACTABLE** attributes for the base key affect the values that these attributes can hold
7182 for the newly-derived key. See the description of each particular key-derivation mechanism in Section
7183 6.42 and 6.43 for any constraints of this type.

7184 If a call to **C_DeriveKey** cannot support the precise template supplied to it, it will fail and return without
7185 creating any key object.

7186 The key object created by a successful call to **C_DeriveKey** will have its **CKA_LOCAL** attribute set to
7187 CK_FALSE. In addition, the object created will have a value for **CKA_UNIQUE_ID** generated and
7188 assigned (See Section 4.4.1).

7189 To partition the derivation keys so they can only derive a subset of keys the attribute
7190 **CKA_DERIVE_TEMPLATE** can be used on the derivation keys to specify an attribute set that will be
7191 added to attributes of the key to be derived. If the attributes do not conflict with the user supplied attribute
7192 template, in 'pTemplate', then the derivation will proceed. The value of this attribute is an attribute
7193 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
7194 attribute is not present on the base derivation keys then no additional attributes will be added. If any
7195 attribute conflict occurs on an attempt to derive a key then the function SHALL return
7196 **CKR_TEMPLATE_INCONSISTENT**.

7197 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
7198 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
7199 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
7200 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
7201 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
7202 CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE,
7203 CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,

7204 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
7205 CKR_PIN_EXPIRED, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
7206 CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
7207 CKR_TOKEN_WRITE_PROTECTED, CKR_USER_NOT_LOGGED_IN.

7208 Example:

```
7209 CK_SESSION_HANDLE hSession;  
7210 CK_OBJECT_HANDLE hPublicKey, hPrivateKey, hKey;  
7211 CK_MECHANISM keyPairMechanism = {  
7212     CKM_DH_PKCS_KEY_PAIR_GEN, NULL_PTR, 0  
7213 };  
7214 CK_BYTE prime[] = {...};  
7215 CK_BYTE base[] = {...};  
7216 CK_BYTE publicKeyValue[128];  
7217 CK_BYTE otherPublicValue[128];  
7218 CK_MECHANISM mechanism = {  
7219     CKM_DH_PKCS_DERIVE, otherPublicValue, sizeof(otherPublicValue)  
7220 };  
7221 CK_ATTRIBUTE template[] = {  
7222     {CKA_VALUE, &publicValue, sizeof(publicValue)}  
7223 };  
7224 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
7225 CK_KEY_TYPE keyType = CKK_DES;  
7226 CK_BBOOL true = CK_TRUE;  
7227 CK_ATTRIBUTE publicKeyTemplate[] = {  
7228     {CKA_PRIME, prime, sizeof(prime)},  
7229     {CKA_BASE, base, sizeof(base)}  
7230 };  
7231 CK_ATTRIBUTE privateKeyTemplate[] = {  
7232     {CKA_DERIVE, &true, sizeof(true)}  
7233 };  
7234 CK_ATTRIBUTE derivedKeyTemplate[] = {  
7235     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
7236     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
7237     {CKA_ENCRYPT, &true, sizeof(true)},  
7238     {CKA_DECRYPT, &true, sizeof(true)}  
7239 };  
7240 CK_RV rv;  
7241  
7242 .  
7243 .  
7244 rv = C_GenerateKeyPair(  
7245     hSession, &keyPairMechanism,  
7246     publicKeyTemplate, 2,
```

```

7247     privateKeyTemplate, 1,
7248     &hPublicKey, &hPrivateKey);
7249 if (rv == CKR_OK) {
7250     rv = C_GetAttributeValue(hSession, hPublicKey, template, 1);
7251     if (rv == CKR_OK) {
7252         /* Put other guy's public value in otherPublicValue */
7253         .
7254         .
7255         rv = C_DeriveKey(
7256             hSession, &mechanism,
7257             hPrivateKey, derivedKeyTemplate, 4, &hKey);
7258         if (rv == CKR_OK) {
7259             .
7260             .
7261         }
7262     }
7263 }

```

5.18.6 C_WrapKeyAuthenticated

```

7265 CK_DECLARE_FUNCTION(CK_RV, C_WrapKeyAuthenticated) (
7266     CK_SESSION_HANDLE hSession,
7267     CK_MECHANISM_PTR pMechanism,
7268     CK_OBJECT_HANDLE hWrappingKey,
7269     CK_OBJECT_HANDLE hKey,
7270     CK_BYTE_PTR pAssociatedData,
7271     CK_ULONG ulAssociatedDataLen,
7272     CK_BYTE_PTR pWrappedKey,
7273     CK_ULONG_PTR pulWrappedKeyLen
7274 );

```

C_WrapKeyAuthenticated wraps (i.e., encrypts) a private or secret key. *hSession* is the session's handle; *pMechanism* points to the wrapping mechanism with the message params; *hWrappingKey* is the handle of the wrapping key; *hKey* is the handle of the key to be wrapped; *pAssociatedData* and *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *pWrappedKey* points to the location that receives the wrapped key; and *pulWrappedKeyLen* points to the location that receives the length of the wrapped key.

C_WrapKeyAuthenticated uses the convention described in section 5.2 on producing output.

The **CKA_WRAP** attribute of the wrapping key, which indicates whether the key supports wrapping, MUST be CK_TRUE. The **CKA_EXTRACTABLE** attribute of the key to be wrapped MUST also be CK_TRUE.

If the key to be wrapped cannot be wrapped for some token-specific reason, despite it having its **CKA_EXTRACTABLE** attribute set to CK_TRUE, then **C_WrapKeyAuthenticated** fails with error code **CKR_KEY_NOT_WRAPPABLE**. If it cannot be wrapped with the specified wrapping key and mechanism solely because of its length, then **C_WrapKeyAuthenticated** fails with error code **CKR_KEY_SIZE_RANGE**.

C_WrapKeyAuthenticated primary use case:

To wrap any secret or private key using an AEAD authenticated mechanism. This allows the ability to use a provider generated (random) IV (GCM) or nonce (CCM) using the standard **CK_*_MESSAGE_PARAMS** with a mechanism as opposed to passing in the IV (GCM) or nonce

7294 (CCM). This IV can then be passed into the mechanism **CK_*_MESSAGE_PARAMS** for
7295 **C_UnwrapKeyAuthenticated**. See section 6.13.3 for further description of AES-GCM authenticated
7296 wrap/unwrap and section 6.13.5 for AES-CCM authenticated wrap/unwrap.

7297 Of course, tokens vary in which types of keys can actually be wrapped with which mechanisms.

7298 To partition the wrapping keys so they can only wrap a subset of extractable keys the attribute
7299 **CKA_WRAP_TEMPLATE** can be used on the wrapping key to specify an attribute set that will be
7300 compared against the attributes of the key to be wrapped. If all attributes match according to the
7301 **C_FindObject** rules of attribute matching, then the wrap will proceed. The value of this attribute is an
7302 attribute template, and the size is the number of items in the template times the size of CK_ATTRIBUTE.
7303 If this attribute is not supplied, then any template is acceptable. If an attribute is not present, it will not be
7304 checked. If any attribute mismatch occurs on an attempt to wrap a key, then the function SHALL return
7305 **CKR_KEY_HANDLE_INVALID**.

7306 Return Values: CKR_ARGUMENTS_BAD, CKR_BUFFER_TOO_SMALL,
7307 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
7308 CKR_DEVICE_REMOVED, CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED,
7309 CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID,
7310 CKR_KEY_NOT_WRAPPABLE, CKR_KEY_SIZE_RANGE, CKR_KEY_UNEXTRACTABLE,
7311 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
7312 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
7313 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN,
7314 CKR_WRAPPING_KEY_HANDLE_INVALID, CKR_WRAPPING_KEY_SIZE_RANGE,
7315 CKR_WRAPPING_KEY_TYPE_INCONSISTENT.

7316 Example:

```
7317 CK_SESSION_HANDLE hSession;  
7318 CK_OBJECT_HANDLE hWrappingKey, hKey;  
7319 CK_BYTE iv[12];  
7320 CK_BYTE tag[16];  
7321 CK_BYTE auth[100];  
7322  
7323 CK_GCM_MESSAGE_PARAMS gcmParams = {  
7324     iv,  
7325     sizeof(iv) * 8,  
7326     96,  
7327     CKG_GENERATE,  
7328     tag,  
7329     sizeof(tag) * 8  
7330 };  
7331 CK_MECHANISM mechanism = {  
7332     CKM_AES_GCM, gcmParams, sizeof(gcmParams)  
7333 };  
7334 CK_BYTE wrappedKey[32]; /* only the wrapped key returned*/  
7335 CK_ULONG ulWrappedKeyLen;  
7336 CK_RV rv;  
7337  
7338 ulWrappedKeyLen = sizeof(wrappedKey);  
7339 rv = C_WrapKeyAuthenticated(  
7340     hSession, &mechanism, hWrappingKey, hKey,
```

```

7341     auth, sizeof(auth),
7342     wrappedKey, &ulWrappedKeyLen);
7343
7344     if (rv == CKR_OK) {
7345         .
7346         .
7347     }

```

7348 5.18.7 C_UnwrapKeyAuthenticated

```

7349 CK_DECLARE_FUNCTION(CK_RV, C_UnwrapKeyAuthenticated) (
7350     CK_SESSION_HANDLE hSession,
7351     CK_MECHANISM_PTR pMechanism,
7352     CK_OBJECT_HANDLE hUnwrappingKey,
7353     CK_BYTE_PTR pWrappedKey,
7354     CK_ULONG ulWrappedKeyLen,
7355     CK_ATTRIBUTE_PTR pTemplate,
7356     CK_ULONG ulAttributeCount,
7357     CK_BYTE_PTR pAssociatedData,
7358     CK_ULONG ulAssociatedDataLen
7359     CK_OBJECT_HANDLE_PTR phKey
7360 );

```

7361 **C_UnwrapKeyAuthenticated** unwraps (*i.e.* decrypts) a wrapped key, creating a new private key or
7362 secret key object. *hSession* is the session's handle; *pMechanism* points to the unwrapping mechanism
7363 with the message params; *hUnwrappingKey* is the handle of the unwrapping key; *pWrappedKey* points to
7364 the wrapped key; *ulWrappedKeyLen* is the length of the wrapped key; *pTemplate* points to the template
7365 for the new key; *ulAttributeCount* is the number of attributes in the template; *pAssociatedData* and
7366 *ulAssociatedDataLen* specify the associated data for an AEAD mechanism; *phKey* points to the location
7367 that receives the handle of the key.

7368 The **CKA_UNWRAP** attribute of the unwrapping key, which indicates whether the key supports
7369 unwrapping, MUST be CK_TRUE.

7370 The new key will have the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, and the
7371 **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE. The **CKA_EXTRACTABLE** attribute is by
7372 default set to CK_TRUE.

7373 Some mechanisms may modify, or attempt to modify, the contents of the *pMechanism* structure at the
7374 same time that the key is unwrapped.

7375 If a call to **C_UnwrapKeyAuthenticated** cannot support the precise template supplied to it, it will fail and
7376 return without creating any key object.

7377 The key object created by a successful call to **C_UnwrapKeyAuthenticated** will have its **CKA_LOCAL**
7378 attribute set to CK_FALSE. In addition, the object created will have a value for **CKA_UNIQUE_ID**
7379 generated and assigned (see section 4.4.1).

7380

7381 **C_UnwrapKeyAuthenticated** Primary use case:

7382 To unwrap any wrapped secret or private key using an AEAD Authenticated mechanism. This
7383 allows the ability to use a provider generated (random) IV (GCM) or nonce (CCM) from
7384 **C_WrapKeyAuthenticated** using the standard **CK_*_MESSAGE_PARAMS**. This IV can then be
7385 passed into the mechanism **CK_*_MESSAGE_PARAMS** for **C_UnwrapKeyAuthenticated**. See
7386 section 6.13.3 for further description of AES-GCM authenticated wrap/unwrap and section 6.13.5
7387 for AES-CCM authenticated wrap/unwrap.

7388 Of course, tokens vary in which types of keys can actually be unwrapped with which mechanisms.

7389 To partition the unwrapping keys so they can only unwrap a subset of keys the attribute
7390 **CKA_UNWRAP_TEMPLATE** can be used on the unwrapping key to specify an attribute set that will be
7391 added to attributes of the key to be unwrapped. If the attributes do not conflict with the user supplied
7392 attribute template in 'pTemplate', then the unwrap will proceed. The value of this attribute is an attribute
7393 template and the size is the number of items in the template times the size of CK_ATTRIBUTE. If this
7394 attribute is not present on the unwrapping key then no additional attributes will be added. If any attribute
7395 conflict occurs on an attempt to unwrap a key then the function SHALL return
7396 **CKR_TEMPLATE_INCONSISTENT**.

7397 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
7398 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
7399 CKR_BUFFER_TOO_SMALL, CKR_CRYPTOKI_NOT_INITIALIZED,
7400 CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
7401 CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID, CKR_FUNCTION_CANCELED,
7402 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY,
7403 CKR_MECHANISM_INVALID, CKR_MECHANISM_PARAM_INVALID, CKR_OK,
7404 CKR_OPERATION_ACTIVE, CKR_PARAMETER_SET_NOT_SUPPORTED, CKR_PIN_EXPIRED,
7405 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY,
7406 CKR_TEMPLATE_INCOMPLETE, CKR_TEMPLATE_INCONSISTENT,
7407 CKR_TOKEN_WRITE_PROTECTED, CKR_UNWRAPPING_KEY_HANDLE_INVALID,
7408 CKR_UNWRAPPING_KEY_SIZE_RANGE, CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT,
7409 CKR_USER_NOT_LOGGED_IN, CKR_WRAPPED_KEY_INVALID,
7410 CKR_WRAPPED_KEY_LEN_RANGE.

7411

7412 Example:

```
7413 CK_SESSION_HANDLE hSession;  
7414 CK_OBJECT_HANDLE hUnwrappingKey, hKey;  
7415 CK_BYTE wrappedKey[32] = {...};  
7416 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;  
7417 CK_KEY_TYPE keyType = CKK_AES;  
7418 CK_BBOOL true = CK_TRUE;  
7419  
7420 CK_ATTRIBUTE template[] = {  
7421     {CKA_CLASS, &keyClass, sizeof(keyClass)},  
7422     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
7423     {CKA_ENCRYPT, &true, sizeof(true)},  
7424     {CKA_DECRYPT, &true, sizeof(true)}  
7425 };  
7426 CK_RV rv;  
7427 CK_BYTE iv[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}; /*value from wrap  
7428 CKG_GENERATE */  
7429 CK_BYTE tag[16];  
7430 CK_BYTE auth[100];  
7431 CK_GCM_MESSAGE_PARAMS gcmParams = {  
7432     iv,  
7433     sizeof(iv) * 8,  
7434     0, /* ignored */  
7435     CKG_NO_GENERATE, /* ignored */  
7436     tag, /* Tag returned from Wrap */
```

```

7437     sizeof(tag) * 8
7438 };
7439 CK_MECHANISM mechanism = {
7440     CKM_AES_GCM, gcmParams, sizeof(gcmParams)
7441 };
7442
7443 .
7444 .
7445 rv = C_UnwrapKeyAuthenticated(
7446     hSession, &mechanism, hUnwrappingKey,
7447     auth, sizeof(auth),
7448     wrappedKey, sizeof(wrappedKey),
7449     template, 4, &hKey);
7450
7451 if (rv == CKR_OK) {
7452     .
7453     .
7454 }

```

7455 5.18.8 C_EncapsulateKey

```

7456 CK_DECLARE_FUNCTION(CK_RV, C_EncapsulateKey) (
7457     CK_SESSION_HANDLE hSession,
7458     CK_MECHANISM_PTR pMechanism,
7459     CK_OBJECT_HANDLE hPublicKey,
7460     CK_ATTRIBUTE_PTR pTemplate,
7461     CK_ULONG ulAttributeCount,
7462     CK_BYTE_PTR pCiphertext,
7463     CK_ULONG_PTR pulCiphertextLen
7464     CK_OBJECT_HANDLE_PTR phKey,
7465 );

```

7466 **C_EncapsulateKey** creates a new secret key object from a public key using a KEM. *hSession* is the
7467 session's handle; *pMechanism* points to a structure that specifies the encapsulation mechanism;
7468 *hPublicKey* is the handle of the public key; *pTemplate* points to the template for the new key;
7469 *ulAttributeCount* is the number of attributes in the template; *pCiphertext* points to the location that
7470 receives the ciphertext needed by decapsulate; *pulCiphertextLen* points to the location to receive the
7471 length; and *phKey* points to the location that receives the handle of the new key. If *pCiphertext* is not big
7472 enough to hold the required ciphertext, *pulCiphertextLen* is set to the space needed to hold *pCiphertext*
7473 and **C_EncapsulateKey** returns **CKR_BUFFER_TOO_SMALL**. The **CKA_ENCAPSULATE** attribute of
7474 the public key, which indicates whether the key supports encapsulation, MUST be CK_TRUE.

7475 The new key will have:

- 7476 • the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE,
- 7477 • the **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE.
- 7478 • the **CKA_EXTRACTABLE** set to the value of the input template with a default of CK_TRUE if
- 7479 not provided,
- 7480 • the **CKA_LOCAL** attribute set to CK_FALSE, and
- 7481 • the **CKA_UNIQUE_ID** attribute generated and assigned per section 4.4.1.

7482 If a call to **C_EncapsulateKey** cannot support the precise template supplied to it, it will fail and return
7483 without creating any key object.

7484

7485 Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY,
 7486 CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID,
 7487 CKR_CRYPTOKI_NOT_INITIALIZED, CKR_CURVE_NOT_SUPPORTED, CKR_DEVICE_ERROR,
 7488 CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED, CKR_DOMAIN_PARAMS_INVALID,
 7489 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
 7490 CKR_HOST_MEMORY, CKR_KEY_HANDLE_INVALID, CKR_KEY_SIZE_RANGE,
 7491 CKR_KEY_TYPE_INCONSISTENT, CKR_MECHANISM_INVALID,
 7492 CKR_MECHANISM_PARAM_INVALID, CKR_OK, CKR_PARAMETER_SET_NOT_SUPPORTED,
 7493 CKR_OPERATION_ACTIVE, CKR_PIN_EXPIRED, CKR_SESSION_CLOSED,
 7494 CKR_SESSION_HANDLE_INVALID, CKR_SESSION_READ_ONLY, CKR_TEMPLATE_INCOMPLETE,
 7495 CKR_TEMPLATE_INCONSISTENT, CKR_TOKEN_WRITE_PROTECTED,
 7496 CKR_USER_NOT_LOGGED_IN.

7497 Example:

```

7498 CK_SESSION_HANDLE hSession;
7499 CK_OBJECT_HANDLE hPublicKey, hKey;
7500 CK_BYTE cipherText[128];
7501 CK_MECHANISM mechanism = {
7502     CKM_ECDH, NULL_PTR, 0
7503 };
7504 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
7505 CK_KEY_TYPE keyType = CKK_AES;
7506 CK_BBOOL true = CK_TRUE;
7507 CK_ATTRIBUTE keyTemplate[] = {
7508     {CKA_CLASS, &keyClass, sizeof(keyClass)},
7509     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7510     {CKA_ENCRYPT, &true, sizeof(true)},
7511     {CKA_DECRYPT, &true, sizeof(true)}
7512 };
7513 CK_RV rv;
7514
7515 .
7516 .
7517 ulCipherTextLen = sizeof(cipherText);
7518 rv = C_EncapsulateKey(
7519     hSession, &mechanism, hPublicKey,
7520     keyTemplate, 4,
7521     cipherText, &ulCipherTextLen,
7522     &hKey);
7523
7524 if (rv == CKR_OK) {
7525     .
7526     .
7527 }
```

5.18.9 C_DecapsulateKey

```
CK_DECLARE_FUNCTION(CK_RV, C_DecapsulateKey) (
    CK_SESSION_HANDLE hSession,
    CK_MECHANISM_PTR pMechanism,
    CK_OBJECT_HANDLE hPrivateKey,
    CK_ATTRIBUTE_PTR pTemplate,
    CK_ULONG ulAttributeCount,
    CK_BYTE_PTR pCiphertext,
    CK_ULONG ulCiphertextLen,
    CK_OBJECT_HANDLE_PTR phKey,
);
```

C_DecapsulateKey creates a new secret key object based on the private key and ciphertext generated by a prior encapsulate operation. This new key (called a 'shared key' in most KEM documentation) is identical to the key returned by **C_EncapsulateKey** when it was called with the matching public key and returned the same ciphertext. This function is a KEM style function. *hSession* is the session's handle; *pMechanism* points to the decapsulation mechanism; *hPrivateKey* is the handle of the decapsulation key; *pTemplate* points to the template for the new key; *ulAttributeCount* is the number of attributes in the template; *pCiphertext* points to the ciphertext in the KEM protocol; *ulCiphertextLen* is the length of the ciphertext; *phKey* points to the location that receives the handle of the decapsulated key.

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

The **CKA_DECAPSULATE** attribute of the private key, which indicates whether the key supports decapsulation, MUST be CK_TRUE.

The new key will have:

- the **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE,
- the **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE.
- the **CKA_EXTRACTABLE** set to the value of the input template with a default of CK_TRUE if not provided,
- the **CKA_LOCAL** attribute set to CK_FALSE, and
- the **CKA_UNIQUE_ID** attribute generated and assigned per section 4.4.1.

Some mechanisms may modify, or attempt to modify, the contents of the pMechanism structure at the same time that the key is decapsulated.

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

Return values: CKR_ARGUMENTS_BAD, CKR_ATTRIBUTE_READ_ONLY, CKR_ATTRIBUTE_TYPE_INVALID, CKR_ATTRIBUTE_VALUE_INVALID, CKR_BUFFER_TOO_SMALL, 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_PARAMETER_SET_NOT_SUPPORTED, 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_UNWRAPPING_KEY_HANDLE_INVALID, CKR_UNWRAPPING_KEY_SIZE_RANGE, CKR_UNWRAPPING_KEY_TYPE_INCONSISTENT, CKR_USER_NOT_LOGGED_IN, CKR_WRAPPED_KEY_INVALID, CKR_WRAPPED_KEY_LEN_RANGE.

Example:

```
CK_SESSION_HANDLE hSession;
```

```

7579 CK_OBJECT_HANDLE hPrivateKey, hKey;
7580 CK_MECHANISM mechanism = {
7581     CKM_ECDH, NULL_PTR, 0
7582 };
7583 CK_BYTE cipherText[35] = {...};
7584 CK_OBJECT_CLASS keyClass = CKO_SECRET_KEY;
7585 CK_KEY_TYPE keyType = CKK_AES;
7586 CK_BBOOL true = CK_TRUE;
7587 CK_ATTRIBUTE template[] = {
7588     {CKA_CLASS, &keyClass, sizeof(keyClass)},
7589     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7590     {CKA_ENCRYPT, &true, sizeof(true)},
7591     {CKA_DECRYPT, &true, sizeof(true)}
7592 };
7593 CK_RV rv;
7594
7595 .
7596 .
7597 rv = C_DecapsulateKey(
7598     hSession, &mechanism, hPrivateKey,
7599     template, 4,
7600     cipherText, sizeof(cipherText),
7601     &hKey);
7602
7603 if (rv == CKR_OK) {
7604     .
7605     .
7606 }

```

5.19 Random number generation functions

Cryptoki provides the following functions for generating random numbers:

5.19.1 C_SeedRandom

```

7610 CK_DECLARE_FUNCTION(CK_RV, C_SeedRandom) (
7611     CK_SESSION_HANDLE hSession,
7612     CK_BYTE_PTR pSeed,
7613     CK_ULONG ulSeedLen
7614 );

```

C_SeedRandom mixes additional seed material into the token's random number generator. *hSession* is the session's handle; *pSeed* points to the seed material; and *ulSeedLen* is the length in bytes of the seed material.

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_ACTIVE, CKR_PENDING,

7622 CKR_RANDOM_SEED_NOT_SUPPORTED, CKR_RANDOM_NO_RNG, CKR_SESSION_CLOSED,
7623 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

7624 Example: see **C_GenerateRandom**.

7625 5.19.2 C_GenerateRandom

```
7626 CK_DECLARE_FUNCTION(CK_RV, C_GenerateRandom) (  
7627     CK_SESSION_HANDLE hSession,  
7628     CK_BYTE_PTR pRandomData,  
7629     CK_ULONG ulRandomLen  
7630 );
```

7631 **C_GenerateRandom** generates random data. *hSession* is the session's handle; *pRandomData* points to
7632 the location that receives the random data; and *ulRandomLen* is the length in bytes of the random data to
7633 be generated.

7634 Return values: CKR_ARGUMENTS_BAD, CKR_CRYPTOKI_NOT_INITIALIZED,
7635 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
7636 CKR_FUNCTION_CANCELED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,
7637 CKR_HOST_MEMORY, CKR_OK, CKR_OPERATION_ACTIVE, CKR_PENDING,
7638 CKR_RANDOM_NO_RNG, CKR_SEED_RANDOM_REQUIRED, CKR_SESSION_CLOSED,
7639 CKR_SESSION_HANDLE_INVALID, CKR_USER_NOT_LOGGED_IN.

7640 Example:

```
7641 CK_SESSION_HANDLE hSession;  
7642 CK_BYTE seed[] = {...};  
7643 CK_BYTE randomData[] = {...};  
7644 CK_RV rv;  
7645  
7646 .  
7647 .  
7648 rv = C_SeedRandom(hSession, seed, sizeof(seed));  
7649 if (rv != CKR_OK) {  
7650     .  
7651     .  
7652 }  
7653 rv = C_GenerateRandom(hSession, randomData, sizeof(randomData));  
7654 if (rv == CKR_OK) {  
7655     .  
7656     .  
7657 }
```

7658 5.20 Parallel function management functions

7659 Cryptoki provides the following functions for managing parallel execution of cryptographic functions.
7660 These functions exist only for backwards compatibility.

7661 5.20.1 C_GetFunctionStatus

```
7662 CK_DECLARE_FUNCTION(CK_RV, C_GetFunctionStatus) (  
7663     CK_SESSION_HANDLE hSession  
7664 );
```

7665 In previous versions of Cryptoki, **C_GetFunctionStatus** obtained the status of a function running in
7666 parallel with an application. Now, however, **C_GetFunctionStatus** is a legacy function which should
7667 simply return the value **CKR_FUNCTION_NOT_PARALLEL**.

7668 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
7669 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
7670 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_SESSION_HANDLE_INVALID,
7671 CKR_SESSION_CLOSED.

7672 5.20.2 C_CancelFunction

```
7673 CK_DECLARE_FUNCTION(CK_RV, C_CancelFunction) (  
7674     CK_SESSION_HANDLE hSession  
7675 );
```

7676 In previous versions of Cryptoki, **C_CancelFunction** cancelled a function running in parallel with an
7677 application. Now, however, **C_CancelFunction** is a legacy function which should simply return the value
7678 **CKR_FUNCTION_NOT_PARALLEL**.

7679 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_FUNCTION_FAILED,
7680 CKR_FUNCTION_NOT_PARALLEL, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
7681 CKR_OPERATION_ACTIVE, CKR_PENDING, CKR_SESSION_HANDLE_INVALID,
7682 CKR_SESSION_CLOSED.

7683 5.21 Asynchronous function management functions

7684 Cryptoki provides the following functions for managing asynchronous execution of cryptographic
7685 functions.

7686 5.21.1 C_AsyncComplete

```
7687 CK_DECLARE_FUNCTION(CK_RV, C_AsyncComplete) (  
7688     CK_SESSION_HANDLE hSession  
7689     CK_UTF8CHAR_PTR pFunctionName,  
7690     CK_ASYNC_DATA_PTR pResult  
7691 );
```

7692 **C_AsyncComplete** checks if the function identified by *pFunctionName* has completed an asynchronous
7693 operation and, if so, returns the associated result(s). *hSession* is the session's handle; *pFunctionName* is
7694 the name of the function whose state is being queried; *pResult* is a pointer to a structure to receive the
7695 result(s) if the function has completed. If the operation has not completed, **CKR_PENDING** is returned,
7696 and the location pointed to by *pResult* is not modified. If **C_AsyncComplete** returns an error other than
7697 **CKR_PENDING**, the *pValue* item of *pResult* is set to the address passed into **C_AsyncJoin** as *pData*, or
7698 the address passed into the original call to *pFunctionName* as the variable-length output buffer, so that it
7699 may be freed if dynamically allocated.

7700 For functions that take a buffer in which to place the result of the operation:

- 7701 ♦ They must return **CKR_BUFFER_TOO_SMALL** if the input buffer is too small. They cannot return
7702 **CKR_PENDING**.
- 7703 ♦ Callers must ensure that the buffer passed into the original function remains available to the module
7704 for the duration of the operation.

7705 Return values: This function's return values are as returned by the function identified by *pFunctionName*.

7706

7707 Example:

```
7708 CK_SESSION_HANDLE hSession;  
7709 CK_OBJECT_HANDLE hKey;  
7710 CK_MECHANISM mechanism = {
```

```

7711     CKM_DES_MAC, NULL_PTR, 0
7712 };
7713
7714 CK_BYTE data[] = {...};
7715 CK_BYTE_PTR mac = NULL_PTR;
7716 CK_ULONG ulMacLen = 0;
7717 CK_RV rv;
7718 .
7719 .
7720 rv = C_SignInit(hSession, &mechanism, hKey);
7721 while (rv == CKR_PENDING)
7722 {
7723     rv = C_AsyncComplete(hSession, (CK_UTF8CHAR_PTR)"C_SignInit", NULL_PTR);
7724     /* rv will contain CKR_PENDING if the operation is still running
7725        or it will contain the return code from the C_SignInit operation */
7726 }
7727
7728 if (rv == CKR_OK)
7729 {
7730     rv = C_SignUpdate(hSession, data, sizeof(data));
7731     while (rv == CKR_PENDING)
7732     {
7733         rv = C_AsyncComplete(hSession, (CK_UTF8CHAR_PTR)"C_SignUpdate",
7734 NULL_PTR);
7735         /* rv will contain CKR_PENDING if the operation is still running
7736            or it will contain the return code from the C_SignUpdate operation */
7737     }
7738     .
7739     .
7740     rv = C_SignFinal(hSession, mac, &ulMacLen);
7741     if (rv == CKR_BUFFER_TOO_SMALL)
7742     {
7743         mac = (CK_BYTE_PTR)malloc(ulMacLen);
7744         rv = C_SignFinal(hSession, mac, &ulMacLen);
7745     }
7746
7747     if (rv == CKR_PENDING)
7748     {
7749         .
7750         .
7751         CK_ASYNC_DATA result;
7752         result.ulVersion = 0;
7753         result.pValue = NULL_PTR;

```

```

7754     result.ulValue = 0;
7755     rv = C_AsyncComplete(hSession, (CK_UTF8CHAR_PTR)"C_SignFinal", &result);
7756     /* on success, result.pValue contains the pointer to the buffer input
7757        to C_SignFinal i.e., on success, mac == result.pValue */
7758 }
7759 }

```

7760 5.21.2 C_AsyncGetID

```

7761 CK_DECLARE_FUNCTION(CK_RV, C_AsyncGetID) (
7762     CK_SESSION_HANDLE hSession
7763     CK_UTF8CHAR_PTR pFunctionName,
7764     CK_ULONG_PTR pulID
7765 );

```

7766 **C_AsyncGetID** is used to persist an operation past a **C_Finalize** call and allow another instance of the
7767 client to reconnect after a call to **C_Initialize**. **C_AsyncGetID** places a module dependent identifier for
7768 the asynchronous operation being performed by the function identified by *pFunctionName*. *hSession* is
7769 the session's handle; *pFunctionName* is the name of the function for which an identifier is being
7770 requested; *pulID* is a pointer to a ULONG to contain the identifier.

7771 An attempt to obtain an identifier for a function that is not performing an asynchronous operation should
7772 fail with the error **CKR_OPERATION_NOT_INITIALIZED**.

7773 An attempt to obtain an identifier for a function that is performing an asynchronous operation, but for
7774 which the module is unable or unwilling to persist past session close should fail with the error

7775 **CKR_STATE_UNSAVEABLE**.

7776 After a successful call to **C_AsyncGetID** the caller should free any memory passed into the original call to
7777 *pFunctionName*.

7778 Return values: This function's return values are CKR_CRYPTOKI_NOT_INITIALIZED,
7779 CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY, CKR_DEVICE_REMOVED,
7780 CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR, CKR_HOST_MEMORY, CKR_OK,
7781 CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID, CKR_ARGUMENTS_BAD,
7782 CKR_TOKEN_NOT_PRESENT, CKR_STATE_UNSAVEABLE.

7783 5.21.3 C_AsyncJoin

```

7784 CK_DECLARE_FUNCTION(CK_RV, C_AsyncJoin) (
7785     CK_SESSION_HANDLE hSession
7786     CK_UTF8CHAR_PTR pFunctionName,
7787     CK_ULONG ulID,
7788     CK_BYTE_PTR pData,
7789     CK_ULONG ulData
7790 );

```

7791 **C_AsyncJoin** checks if the function identified by *pFunctionName* and *ulID* is a valid asynchronous
7792 operation and, if so, reconnects the client application to the module using the buffer specified by *pData*
7793 and *ulData* in place of those passed into the original call to *pFunctionName*. *hSession* is the session's
7794 handle; *pFunctionName* is the name of the function to join; *ulID* is an identifier returned by
7795 **C_AsyncGetID**; *pData* is a pointer to a buffer to contain the result once the function has completed;
7796 *ulData* is the length in bytes of the data.

7797 This function follows the conventions in section 5.2 with respect to *pData* and *ulData* if the function
7798 identified by *pFunctionName* requires an output buffer. If the function identified by *pFunctionName* does
7799 not require an output buffer *pData* and *ulData* should be ignored.

7800 Return values: CKR_CRYPTOKI_NOT_INITIALIZED, CKR_DEVICE_ERROR, CKR_DEVICE_MEMORY,
7801 CKR_DEVICE_REMOVED, CKR_FUNCTION_FAILED, CKR_GENERAL_ERROR,

7802 CKR_HOST_MEMORY, CKR_OK, CKR_SESSION_CLOSED, CKR_SESSION_HANDLE_INVALID,
7803 CKR_ARGUMENTS_BAD, CKR_TOKEN_NOT_PRESENT, CKR_SAVED_STATE_INVALID,
7804 CKR_BUFFER_TOO_SMALL.

7805 5.22 Callback functions

7806 Cryptoki sessions can use function pointers of type **CK_NOTIFY** to notify the application of certain
7807 events.

7808 5.22.1 Surrender callbacks

7809 Cryptographic functions (*i.e.*, any functions falling under one of these categories: encryption functions;
7810 decryption functions; message digesting functions; signing and MACing functions; functions for verifying
7811 signatures and MACs; dual-purpose cryptographic functions; key management functions; random number
7812 generation functions) executing in Cryptoki sessions can periodically surrender control to the application
7813 who called them if the session they are executing in had a notification callback function associated with it
7814 when it was opened. They do this by calling the session's callback with the arguments (hSession,
7815 **CKN_SURRENDER**, pApplication), where hSession is the session's handle and pApplication was
7816 supplied to **C_OpenSession** when the session was opened. Surrender callbacks should return either the
7817 value **CKR_OK** (to indicate that Cryptoki should continue executing the function) or the value
7818 **CKR_CANCEL** (to indicate that Cryptoki should abort execution of the function). Of course, before
7819 returning one of these values, the callback function can perform some computation, if desired.

7820 A typical use of a surrender callback might be to give an application user feedback during a lengthy key
7821 pair generation operation. Each time the application receives a callback, it could display an additional "."
7822 to the user. It might also examine the keyboard's activity since the last surrender callback, and abort the
7823 key pair generation operation (probably by returning the value **CKR_CANCEL**) if the user hit <ESCAPE>.

7824 A Cryptoki library is not *required* to make *any* surrender callbacks.

7825 5.22.2 Vendor-defined callbacks

7826 Library vendors can also define additional types of callbacks. Because of this extension capability,
7827 application-supplied notification callback routines should examine each callback they receive, and if they
7828 are unfamiliar with the type of that callback, they should immediately give control back to the library by
7829 returning with the value **CKR_OK**.

6 Mechanisms

6.1 RSA

Table 36, Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_RSA_PKCS_KEY_PAIR_GEN					✓			
CKM_RSA_X9_31_KEY_PAIR_GEN					✓			
CKM_RSA_PKCS	✓ ¹	✓ ¹	✓			✓		✓
CKM_RSA_PKCS_OAEP	✓ ¹					✓		✓
CKM_RSA_PKCS_PSS		✓ ¹						
CKM_RSA_9796		✓ ¹	✓					
CKM_RSA_X_509	✓ ¹	✓ ¹	✓			✓		✓
CKM_RSA_X9_31		✓ ¹						
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	✓ ¹					✓		
CKM_RSA_PKCS_OAEP_TPM_1_1	✓ ¹					✓		
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		✓						

¹ Single-part operations only

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

7840	CKM_RSA_9796
7841	CKM_RSA_X_509
7842	CKM_MD2_RSA_PKCS
7843	CKM_MD5_RSA_PKCS
7844	CKM_SHA1_RSA_PKCS
7845	CKM_SHA224_RSA_PKCS
7846	CKM_SHA256_RSA_PKCS
7847	CKM_SHA384_RSA_PKCS
7848	CKM_SHA512_RSA_PKCS
7849	CKM_RIPEMD128_RSA_PKCS
7850	CKM_RIPEMD160_RSA_PKCS
7851	CKM_RSA_PKCS_OAEP
7852	CKM_RSA_X9_31_KEY_PAIR_GEN
7853	CKM_RSA_X9_31
7854	CKM_SHA1_RSA_X9_31
7855	CKM_RSA_PKCS_PSS
7856	CKM_SHA1_RSA_PKCS_PSS
7857	CKM_SHA224_RSA_PKCS_PSS
7858	CKM_SHA256_RSA_PKCS_PSS
7859	CKM_SHA512_RSA_PKCS_PSS
7860	CKM_SHA384_RSA_PKCS_PSS
7861	CKM_RSA_PKCS_TPM_1_1
7862	CKM_RSA_PKCS_OAEP_TPM_1_1
7863	CKM_RSA_AES_KEY_WRAP
7864	CKM_SHA3_224_RSA_PKCS
7865	CKM_SHA3_256_RSA_PKCS
7866	CKM_SHA3_384_RSA_PKCS
7867	CKM_SHA3_512_RSA_PKCS
7868	CKM_SHA3_224_RSA_PKCS_PSS
7869	CKM_SHA3_256_RSA_PKCS_PSS
7870	CKM_SHA3_384_RSA_PKCS_PSS
7871	CKM_SHA3_512_RSA_PKCS_PSS
7872	

7873 6.1.2 RSA public key objects

7874 RSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_RSA**) hold RSA public keys.
7875 The following table defines the RSA public key object attributes, in addition to the common attributes
7876 defined for this object class:

7877 Table 37, 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
CKA_PUBLIC_CRC64_VALUE ^{1,4,13}	Byte array	The CRC-64-ECMA calculated over the CKA_MODULUS attribute

7878 Refer to Table 13 for footnotes

7879 Depending on the token, there may be limits on the length of key components. See [PKCS #1] for more
7880 information on RSA keys.

7881 The following is a sample template for creating an RSA public key object:

```
7882 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
7883 CK_KEY_TYPE keyType = CKK_RSA;
7884 CK_UTF8CHAR label[] = "An RSA public key object";
7885 CK_BYTE modulus[] = {...};
7886 CK_BYTE exponent[] = {...};
7887 CK_BBOOL true = CK_TRUE;
7888 CK_ATTRIBUTE template[] = {
7889     {CKA_CLASS, &class, sizeof(class)},
7890     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7891     {CKA_TOKEN, &true, sizeof(true)},
7892     {CKA_LABEL, label, sizeof(label)-1},
7893     {CKA_WRAP, &true, sizeof(true)},
7894     {CKA_ENCRYPT, &true, sizeof(true)},
7895     {CKA_MODULUS, modulus, sizeof(modulus)},
7896     {CKA_PUBLIC_EXPONENT, exponent, sizeof(exponent)}
7897 };
```

7898 **6.1.3 RSA private key objects**

7899 RSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_RSA**) hold RSA private keys.
7900 The following table defines the RSA private key object attributes, in addition to the common attributes
7901 defined for this object class:

7902 Table 38, 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 $q^{-1} \bmod p$
CKA_PUBLIC_CRC64_VALUE ^{1,4,13}	Byte array	The CRC-64-ECMA calculated over the CKA_MODULUS attribute

7903 Refer to Table 13 for footnotes

7904 Depending on the token, there may be limits on the length of the key components. See [PKCS #1] for
7905 more information on RSA keys.

7906 Tokens vary in what they actually store for RSA private keys. Some tokens store all of the above
7907 attributes, which can assist in performing rapid RSA computations. Other tokens might store only the
7908 **CKA_MODULUS** and **CKA_PRIVATE_EXPONENT** values. Effective with version 2.40, tokens **MUST**
7909 also store **CKA_PUBLIC_EXPONENT**. This permits the retrieval of sufficient data to reconstitute the
7910 associated public key.

7911 Because of this, Cryptoki is flexible in dealing with RSA private key objects. When a token generates an
7912 RSA private key, it stores whichever of the fields in Table 38 it keeps track of. Later, if an application asks
7913 for the values of the key's various attributes, Cryptoki supplies values only for attributes whose values it
7914 can obtain (*i.e.*, if Cryptoki is asked for the value of an attribute it cannot obtain, the request fails). Note
7915 that a Cryptoki implementation may or may not be able and/or willing to supply various attributes of RSA
7916 private keys which are not actually stored on the token. *E.g.*, if a particular token stores values only for
7917 the **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, and **CKA_PRIME_2** attributes, then Cryptoki is
7918 certainly *able* to report values for all the attributes above (since they can all be computed efficiently from
7919 these three values). However, a Cryptoki implementation may or may not actually do this extra
7920 computation. The only attributes from Table 38 for which a Cryptoki implementation is *required* to be able
7921 to return values are **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT** and **CKA_PRIVATE_EXPONENT**. A
7922 token **SHOULD** also be able to return **CKA_PUBLIC_KEY_INFO** for an RSA private key.

7923 If an RSA private key object is created on a token, and more attributes from Table 38 are supplied to the
7924 object creation call than are supported by the token, the extra attributes are likely to be thrown away. If an
7925 attempt is made to create an RSA private key object on a token with insufficient attributes for that
7926 particular token, then the object creation call fails and returns **CKR_TEMPLATE_INCOMPLETE**.

7927 Note that when generating an RSA private key, there is no **CKA_MODULUS_BITS** attribute specified.
7928 This is because RSA private keys are only generated as part of an RSA key *pair*, and the
7929 **CKA_MODULUS_BITS** attribute for the pair is specified in the template for the RSA public key.

7930 The following is a sample template for creating an RSA private key object:

```

7931     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
7932     CK_KEY_TYPE keyType = CKK_RSA;
7933     CK_UTF8CHAR label[] = "An RSA private key object";
7934     CK_BYTE subject[] = {...};
7935     CK_BYTE id[] = {123};
7936     CK_BYTE modulus[] = {...};
7937     CK_BYTE publicExponent[] = {...};
7938     CK_BYTE privateExponent[] = {...};
7939     CK_BYTE prime1[] = {...};
7940     CK_BYTE prime2[] = {...};
7941     CK_BYTE exponent1[] = {...};
7942     CK_BYTE exponent2[] = {...};
7943     CK_BYTE coefficient[] = {...};
7944     CK_BBOOL true = CK_TRUE;
7945     CK_ATTRIBUTE template[] = {
7946         {CKA_CLASS, &class, sizeof(class)},
7947         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
7948         {CKA_TOKEN, &true, sizeof(true)},
7949         {CKA_LABEL, label, sizeof(label)-1},
7950         {CKA_SUBJECT, subject, sizeof(subject)},
7951         {CKA_ID, id, sizeof(id)},
7952         {CKA_SENSITIVE, &true, sizeof(true)},
7953         {CKA_DECRYPT, &true, sizeof(true)},

```

```

7954     {CKA_SIGN, &true, sizeof(true)},
7955     {CKA_MODULUS, modulus, sizeof(modulus)},
7956     {CKA_PUBLIC_EXPONENT, publicExponent,
7957      sizeof(publicExponent)},
7958     {CKA_PRIVATE_EXPONENT, privateExponent,
7959      sizeof(privateExponent)},
7960     {CKA_PRIME_1, prime1, sizeof(prime1)},
7961     {CKA_PRIME_2, prime2, sizeof(prime2)},
7962     {CKA_EXPONENT_1, exponent1, sizeof(exponent1)},
7963     {CKA_EXPONENT_2, exponent2, sizeof(exponent2)},
7964     {CKA_COEFFICIENT, coefficient, sizeof(coefficient)}
7965 };

```

7966 6.1.4 PKCS #1 RSA key pair generation

7967 The PKCS #1 RSA key pair generation mechanism, denoted **CKM_RSA_PKCS_KEY_PAIR_GEN**, is a
7968 key pair generation mechanism based on the RSA public-key cryptosystem, as defined in [PKCS #1].

7969 It does not have a parameter.

7970 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
7971 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
7972 template for the public key. The **CKA_PUBLIC_EXPONENT** may be omitted in which case the
7973 mechanism shall supply the public exponent attribute using the default value of 0x10001 (65537). Specific
7974 implementations may use a random value or an alternative default if 0x10001 cannot be used by the
7975 token.

7976 Note: Implementations strictly compliant with version 2.11 or prior versions may generate an error if this
7977 attribute is omitted from the template. Experience has shown that many implementations of 2.11 and prior
7978 did allow the **CKA_PUBLIC_EXPONENT** attribute to be omitted from the template, and behaved as
7979 described above. The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**,
7980 and **CKA_PUBLIC_EXPONENT** attributes to the new public key. **CKA_PUBLIC_EXPONENT** will be
7981 copied from the template if supplied. **CKR_TEMPLATE_INCONSISTENT** shall be returned if the
7982 implementation cannot use the supplied exponent value. It contributes the **CKA_CLASS** and
7983 **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes
7984 to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**,
7985 **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**.
7986 Other attributes supported by the RSA public and private key types (specifically, the flags indicating which
7987 functions the keys support) may also be specified in the templates for the keys, or else are assigned
7988 default initial values.

7989 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
7990 specify the supported range of RSA modulus sizes, in bits.

7991 6.1.5 X9.31 RSA key pair generation

7992 The X9.31 RSA key pair generation mechanism, denoted **CKM_RSA_X9_31_KEY_PAIR_GEN**, is a key
7993 pair generation mechanism based on the RSA public-key cryptosystem, as defined in X9.31.

7994 It does not have a parameter.

7995 The mechanism generates RSA public/private key pairs with a particular modulus length in bits and public
7996 exponent, as specified in the **CKA_MODULUS_BITS** and **CKA_PUBLIC_EXPONENT** attributes of the
7997 template for the public key.

7998 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_MODULUS**, and
7999 **CKA_PUBLIC_EXPONENT** attributes to the new public key. It contributes the **CKA_CLASS** and
8000 **CKA_KEY_TYPE** attributes to the new private key; it may also contribute some of the following attributes
8001 to the new private key: **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**,
8002 **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, **CKA_COEFFICIENT**.

Other attributes supported by the RSA 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. Unlike the **CKM_RSA_PKCS_KEY_PAIR_GEN** mechanism, this mechanism is guaranteed to generate p and q values, **CKA_PRIME_1** and **CKA_PRIME_2** respectively, that meet the strong primes requirement of X9.31.

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

6.1.6 PKCS #1 v1.5 RSA

The PKCS #1 v1.5 RSA mechanism, denoted **CKM_RSA_PKCS**, is a multi-purpose mechanism based 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; and key encapsulation and key decapsulation. 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.

When the mechanism is used in key encapsulation, the secret key is generated in the **C_EncapsulateKey** function and then wrapped with RSA PKCS #1 v1.5. **C_DecapsulateKey** is exactly equivalent to **C_UnwrapKey** for RSA PKCS.

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 39, 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 ¹	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 ¹	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
C_EncapsulateKey	RSA public key	none	$\leq k-11, k$	block type 02
C_DecapsulateKey	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 40, 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

CK_RSA_PKCS_MGF_TYPE_PTR is a pointer to a **CK_RSA_PKCS_MGF_TYPE**.

♦ CK_RSA_PKCS_OAEP_SOURCE_TYPE; CK_RSA_PKCS_OAEP_SOURCE_TYPE_PTR

CK_RSA_PKCS_OAEP_SOURCE_TYPE is used to indicate the source of the encoding parameter when formatting a message block for the PKCS #1 OAEP encryption scheme. It is defined as follows:

```
typedef CK_ULONG CK_RSA_PKCS_OAEP_SOURCE_TYPE;
```

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

Table 41, PKCS #1 RSA OAEP: Encoding parameter sources

Source Identifier	Value
CKZ_DATA_SPECIFIED	0x00000001UL

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

♦ CK_RSA_PKCS_OAEP_PARAMS; CK_RSA_PKCS_OAEP_PARAMS_PTR

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

```
typedef struct CK_RSA_PKCS_OAEP_PARAMS {  
    CK_MECHANISM_TYPE    hashAlg;
```

```

8065         CK_RSA_PKCS_MGF_TYPE          mgf;
8066         CK_RSA_PKCS_OAEP_SOURCE_TYPE   source;
8067         CK_VOID_PTR                     pSourceData;
8068         CK_ULONG                        ulSourceDataLen;
8069     } CK_RSA_PKCS_OAEP_PARAMS;

```

8070

8071 The fields of the structure have the following meanings:

8072	hashAlg	mechanism ID of the message digest algorithm used to calculate
8073		the digest of the encoding parameter
8074	mgf	mask generation function to use on the encoded block
8075	source	must be CKZ_DATA_SPECIFIED
8076	pSourceData	pointer to the optional label L to be associated with the message; it
8077		must be NULL_PTR if the caller wants the default label (the empty
8078		string) to be used (as per [RFC 8017])
8079	ulSourceDataLen	length in bytes of the data pointed to by pSourceData; it must be 0 if
8080		the caller wants the default label (the empty string) to be used (as
8081		per [RFC 8017])

8082 **CK_RSA_PKCS_OAEP_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_OAEP_PARAMS**.

8083

8084 6.1.8 PKCS #1 RSA OAEP

8085 The PKCS #1 RSA OAEP mechanism, denoted **CKM_RSA_PKCS_OAEP**, is a multi-purpose
8086 mechanism based on the RSA public-key cryptosystem and the OAEP block format defined in [PKCS #1].
8087 It supports single-part encryption and decryption; key wrapping and key unwrapping; and key
8088 encapsulation and key decapsulation.

8089 It has a parameter, a **CK_RSA_PKCS_OAEP_PARAMS** structure.

8090 This mechanism can wrap and unwrap any secret key of appropriate length. Of course, a particular token
8091 may not be able to wrap/unwrap every appropriate-length secret key that it supports. For wrapping, the
8092 “input” to the encryption operation is the value of the **CKA_VALUE** attribute of the key that is wrapped;
8093 similarly for unwrapping. The mechanism does not wrap the key type or any other information about the
8094 key, except the key length; the application must convey these separately. In particular, the mechanism
8095 contributes only the **CKA_CLASS** and **CKA_VALUE** (and **CKA_VALUE_LEN**, if the key has it) attributes
8096 to the recovered key during unwrapping; other attributes must be specified in the template.

8097 When the mechanism is used in key encapsulation, the secret key is generated in the
8098 **C_EncapsulateKey** function and then wrapped with RSA OAEP. **C_DecapsulateKey** is exactly
8099 equivalent to **C_UnwrapKey** for RSA OAEP.

8100 Constraints on key types and the length of the data are summarized in the following table. For encryption
8101 and decryption, the input and output data may begin at the same location in memory. In the table, *k* is the
8102 length in bytes of the RSA modulus, and *hLen* is the output length of the message digest algorithm
8103 specified by the *hashAlg* field of the **CK_RSA_PKCS_OAEP_PARAMS** structure.

8104 Table 42, PKCS #1 RSA OAEP: Key And Data Length

Function	Key type	Input length	Output length
C_Encrypt ¹	RSA public key	$\leq k-2-2hLen$	k
C_Decrypt ¹	RSA private key	k	$\leq k-2-2hLen$
C_WrapKey	RSA public key	$\leq k-2-2hLen$	k
C_UnwrapKey	RSA private key	k	$\leq k-2-2hLen$
C_EncapsulateKey	RSA public key	none	$\leq k-2-2hLen, k$
C_DecapsulateKey	RSA private key	k	$\leq k-2-2hLen$

8105 ¹ Single-part operations only.

8106 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8107 specify the supported range of RSA modulus sizes, in bits.

8108 **6.1.9 PKCS #1 RSA PSS mechanism parameters**

8109 ♦ **CK_RSA_PKCS_PSS_PARAMS;**
8110 **CK_RSA_PKCS_PSS_PARAMS_PTR**

8111 **CK_RSA_PKCS_PSS_PARAMS** is a structure that provides the parameters to the
8112 **CKM_RSA_PKCS_PSS** mechanism. The structure is defined as follows:

```
8113     typedef struct CK_RSA_PKCS_PSS_PARAMS {  
8114         CK_MECHANISM_TYPE      hashAlg;  
8115         CK_RSA_PKCS_MGF_TYPE   mgf;  
8116         CK_ULONG                sLen;  
8117     } CK_RSA_PKCS_PSS_PARAMS;
```

8118
8119 The fields of the structure have the following meanings:

8120	hashAlg	hash algorithm used in the PSS encoding; if the signature
8121		mechanism does not include message hashing, then this value must
8122		be the mechanism used by the application to generate the message
8123		hash; if the signature mechanism includes hashing, then this value
8124		must match the hash algorithm indicated by the signature
8125		mechanism
8126	mgf	mask generation function to use on the encoded block
8127	sLen	length, in bytes, of the salt value used in the PSS encoding; typical
8128		values are the length of the message hash and zero

8129 **CK_RSA_PKCS_PSS_PARAMS_PTR** is a pointer to a **CK_RSA_PKCS_PSS_PARAMS**.

8130 **6.1.10 PKCS #1 RSA PSS**

8131 The PKCS #1 RSA PSS mechanism, denoted **CKM_RSA_PKCS_PSS**, is a mechanism based on the
8132 RSA public-key cryptosystem and the PSS block format defined in [PKCS #1]. It supports single-part
8133 signature generation and verification without message recovery. This mechanism corresponds only to the
8134 part of [PKCS #1] that involves block formatting and RSA, given a hash value; it does not compute a hash
8135 value on the message to be signed.

8136 It has a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must be less than or
8137 equal to $k^*-2-hLen$ and *hLen* is the length of the input to the **C_Sign** or **C_Verify** function. k^* is the length
8138 in bytes of the RSA modulus, except if the length in bits of the RSA modulus is one more than a multiple
8139 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.

Table 43, PKCS #1 RSA PSS: Key And Data Length

Function	Key type	Input length	Output length
C_Sign ¹	RSA private key	$hLen$	k
C_Verify ¹	RSA public key	$hLen, k$	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.

6.1.11 ISO/IEC 9796 RSA

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

This mechanism processes only byte strings, whereas ISO/IEC 9796 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.

Table 44, 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 ¹	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 ¹	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; and key encapsulation and key decapsulation. 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.

This mechanism can encapsulate and decapsulate keys according to RSASVE defined in [FIPS SP 800-56B].

For all operations other than encapsulate, 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 45, 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 ¹	RSA private key	$\leq k$	k
C_Verify ¹	RSA public key	$\leq k, k^2$	N/A
C_VerifyRecover ¹	RSA public key	k	k
C_WrapKey	RSA public key	$\leq k$	k
C_UnwrapKey	RSA private key	k	$\leq k$ (specified in template)
C_EncapsulateKey	RSA public key	none	k, k
C_DecapsulateKey	RSA private key	k	k

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 46, 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 hashing

The PKCS #1 v1.5 RSA signature with hashing, denoted **CKM_<hash>_RSA_PKCS** where <hash> identifies a hash function as per Table 47, 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 as per Table 47, and as in the scheme RSASSA-PKCS1-v1_5 in the current version of PKCS #1, where the underlying hash function is the hash function as per Table 47.

8243 Table 47, PKCS #1 v1.5 RSA signature with hashing: mechanisms and hash functions

Mechanism	Hash function	Hash value length in bytes	Object identifier
CKM_MD2_RSA_PKCS	MD2	16	md2WithRSAEncryption
CKM_MD5_RSA_PKCS	MD5	16	md5WithRSAEncryption
CKM_RIPEMD128_RSA_PKCS	RIPEMD-128	16	ripemd128WithRSAEncryption
CKM_RIPEMD160_RSA_PKCS	RIPEMD-160	20	ripemd160WithRSAEncryption
CKM_SHA1_RSA_PKCS	SHA-1	20	sha1WithRSAEncryption
CKM_SHA224_RSA_PKCS	SHA-224	28	sha224WithRSAEncryption
CKM_SHA256_RSA_PKCS	SHA-256	32	sha256WithRSAEncryption
CKM_SHA384_RSA_PKCS	SHA-384	48	sha384WithRSAEncryption
CKM_SHA512_RSA_PKCS	SHA-512	64	sha512WithRSAEncryption
CKM_SHA3_224_RSA_PKCS	SHA3-224	28	id-rsassa-pkcs1-v1-5-with-sha3-224
CKM_SHA3_256_RSA_PKCS	SHA3-256	32	id-rsassa-pkcs1-v1-5-with-sha3-256
CKM_SHA3_384_RSA_PKCS	SHA3-384	48	id-rsassa-pkcs1-v1-5-with-sha3-384
CKM_SHA3_512_RSA_PKCS	SHA3-512	64	id-rsassa-pkcs1-v1-5-with-sha3-512

8244 Note: **CKM_MD2_RSA_PKCS**, **CKM_MD5_RSA_PKCS**, **CKM_RIPEMD128_RSA_PKCS** and
8245 **CKM_RIPEMD160_RSA_PKCS** are deprecated with PKCS #11 3.20. New implementations shall not use
8246 these mechanisms anymore.

8247 None of these mechanisms has a parameter.

8248 Constraints on key types and the length of the data for these mechanisms are summarized in the
8249 following table. In the table, k is the length in bytes of the RSA modulus. For the PKCS #1 v1.5 RSA
8250 signature with hashing mechanisms, k must be at least 11 bytes more than the length of the hash value
8251 as indicated in Table 47.

8252 Table 48, 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

8253 2 Data length, signature length.

8254 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
8255 structure specify the supported range of RSA modulus sizes, in bits.

8256 **6.1.15 PKCS #1 RSA PSS signature with hashing**

8257 The PKCS #1 RSA PSS signature with hashing, denoted **CKM_<hash>_RSA_PKCS_PSS** where
8258 <hash> identifies a hash function as per Table 49, performs single- and multiple-part digital signatures
8259 and verification operations without message recovery. The operations performed are as described in
8260 [PKCS #1] with the object identifier id-RSASSA-PSS, i.e., as in the scheme RSASSA-PSS in [PKCS #1]
8261 where the underlying hash function is the hash function as per Table 49.

8262 Table 49, PKCS #1 RSA PSS signature with hashing: mechanisms and hash functions

Mechanism	Hash function	Hash value length in bytes
CKM_SHA1_RSA_PKCS_PSS	SHA-1	20
CKM_SHA224_RSA_PKCS_PSS	SHA-224	28
CKM_SHA256_RSA_PKCS_PSS	SHA-256	32
CKM_SHA384_RSA_PKCS_PSS	SHA-384	48
CKM_SHA512_RSA_PKCS_PSS	SHA-512	64
CKM_SHA3_224_RSA_PKCS_PSS	SHA3-224	28
CKM_SHA3_256_RSA_PKCS_PSS	SHA3-256	32
CKM_SHA3_384_RSA_PKCS_PSS	SHA3-384	48
CKM_SHA3_512_RSA_PKCS_PSS	SHA3-512	64

8263

8264 The mechanisms have a parameter, a **CK_RSA_PKCS_PSS_PARAMS** structure. The *sLen* field must

8265 be less than or equal to $k^* - 2 \cdot hLen$ where *hLen* is the length in bytes of the hash value as indicated in

8266 Table 49. k^* is the length in bytes of the RSA modulus, except if the length in bits of the RSA modulus is

8267 one more than a multiple of 8, in which case k^* is one less than the length in bytes of the RSA modulus.

8268 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*

8269 is the length in bytes of the RSA modulus.

8270 Table 50, 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

8271 2 Data length, signature length.

8272 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure

8273 specify the supported range of RSA modulus sizes, in bits.

8274 **6.1.16 ANSI X9.31 RSA signature with SHA-1**

8275 The ANSI X9.31 RSA signature with SHA-1 mechanism, denoted **CKM_SHA1_RSA_X9_31**, performs

8276 single- and multiple-part digital signatures and verification operations without message recovery. The

8277 operations performed are as described in [ANSI X9.31].

8278 This mechanism does not have a parameter.

8279 Constraints on key types and the length of the data for these mechanisms are summarized in the

8280 following table. In the table, *k* is the length in bytes of the RSA modulus. For all operations, the *k* value

8281 must be at least 128 and a multiple of 32 as specified in [ANSI X9.31].

8282 Table 51, 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

8283 2 Data length, signature length.

8284 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**

8285 structure specify the supported range of RSA modulus sizes, in bits.

6.1.17 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 52, TPM 1.1b and TPM 1.2 PKCS #1 v1.5 RSA: Key And Data Length

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

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.

6.1.18 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 53, 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	$\leq k-2-40-5$	<i>k</i>
C_Decrypt ¹	RSA private key	<i>k</i>	$\leq k-2-40-5$
C_WrapKey	RSA public key	$\leq k-2-40-5$	<i>k</i>
C_UnwrapKey	RSA private key	<i>k</i>	$\leq k-2-40-5$

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.

6.1.19 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*.

- 8370 • Un-wraps the target key from the second part with the temporary AES key using
8371 **CKM_AES_KEY_WRAP_KWP**.
- 8372 • Zeroizes the temporary AES key.
- 8373 • Returns the handle to the newly unwrapped target key.

8374 Table 54, CKM_RSA_AES_KEY_WRAP Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_RSA_AES_KEY_WRAP						✓		

8375 6.1.20 RSA AES KEY WRAP mechanism parameters

- 8376 ♦ **CK_RSA_AES_KEY_WRAP_PARAMS;**
8377 **CK_RSA_AES_KEY_WRAP_PARAMS_PTR**

8378 **CK_RSA_AES_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
8379 **CKM_RSA_AES_KEY_WRAP** mechanism. It is defined as follows:

```
8380 typedef struct CK_RSA_AES_KEY_WRAP_PARAMS {
8381     CK_ULONG                ulAESKeyBits;
8382     CK_RSA_PKCS_OAEP_PARAMS_PTR pOAEPParams;
8383 } CK_RSA_AES_KEY_WRAP_PARAMS;
```

8384 The fields of the structure have the following meanings:

8385 ulAESKeyBits length of the temporary AES key in bits. Can be only 128, 192 or
8386 256.

8387 pOAEPParams pointer to the parameters of the temporary AES key wrapping. See
8388 also the description of PKCS #1 RSA OAEP mechanism
8389 parameters.

8390 **CK_RSA_AES_KEY_WRAP_PARAMS_PTR** is a pointer to a **CK_RSA_AES_KEY_WRAP_PARAMS**.

8391 6.1.21 FIPS 186-4

8392 When **CKM_RSA_PKCS** is operated in FIPS mode, the length of the modulus SHALL only be 1024,
8393 2048, or 3072 bits.

8394 6.2 DSA

8395 Table 55, DSA Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_DSA_KEY_PAIR_GEN					✓			
CKM_DSA_PARAMETER_GEN					✓			

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_DSA_PROBABILISTIC_PARAMETER_GEN					✓			
CKM_DSA_SHAWA_TAYLOR_PARAMETER_GEN					✓			
CKM_DSA_FIPS_G_GEN					✓			
CKM_DSA		✓ ¹						
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		✓						

8396 1 Single-part operations only

8397 6.2.1 Definitions

8398 This section defines the key type “**CKK_DSA**” for type CK_KEY_TYPE as used in the **CKA_KEY_TYPE**
8399 attribute of DSA key objects.

8400 Mechanisms:

- 8401 CKM_DSA_KEY_PAIR_GEN
- 8402 CKM_DSA
- 8403 CKM_DSA_SHA1
- 8404 CKM_DSA_SHA224
- 8405 CKM_DSA_SHA256
- 8406 CKM_DSA_SHA384
- 8407 CKM_DSA_SHA512
- 8408 CKM_DSA_SHA3_224
- 8409 CKM_DSA_SHA3_256
- 8410 CKM_DSA_SHA3_384
- 8411 CKM_DSA_SHA3_512
- 8412 CKM_DSA_PARAMETER_GEN
- 8413 CKM_DSA_PROBABILISTIC_PARAMETER_GEN
- 8414 CKM_DSA_SHAWA_TAYLOR_PARAMETER_GEN
- 8415 CKM_DSA_FIPS_G_GEN
- 8416

8417 ♦ **CK_DSA_PARAMETER_GEN_PARAM**

8418 CK_DSA_PARAMETER_GEN_PARAM is a structure which provides and returns parameters for the
8419 [NIST FIPS 186-4] parameter generating algorithms.

8420 CK_DSA_PARAMETER_GEN_PARAM_PTR is a pointer to a CK_DSA_PARAMETER_GEN_PARAM.

8421
8422 typedef struct CK_DSA_PARAMETER_GEN_PARAM {
8423 CK_MECHANISM_TYPE hash;
8424 CK_BYTE_PTR pSeed;
8425 CK_ULONG ulSeedLen;
8426 CK_ULONG ulIndex;
8427 } CK_DSA_PARAMETER_GEN_PARAM;

8428
8429 The fields of the structure have the following meanings:

8430	hash	Mechanism value for the base hash used in PQG generation, Valid
8431		values are CKM_SHA_1 , CKM_SHA224 , CKM_SHA256 ,
8432		CKM_SHA384 , CKM_SHA512 .
8433	pSeed	Seed value used to generate PQ and G. This value is returned by
8434		CKM_DSA_PROBABILISTIC_PARAMETER_GEN ,
8435		CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN , and passed
8436		into CKM_DSA_FIPS_G_GEN .
8437	ulSeedLen	Length of seed value.
8438	ulIndex	Index value for generating G. Input for CKM_DSA_FIPS_G_GEN .
8439		Ignored by CKM_DSA_PROBABILISTIC_PARAMETER_GEN and
8440		CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN .

8441 **6.2.2 DSA public key objects**

8442 DSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_DSA**) hold DSA public keys.
8443 The following table defines the DSA public key object attributes, in addition to the common attributes
8444 defined for this object class:

8445 *Table 56, DSA Public Key Object Attributes*

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

8446 ¹ Refer to Table 13 for footnotes

8447 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
8448 parameters”. See [FIPS PUB 186-4] for more information on DSA keys.

8449 The following is a sample template for creating a DSA public key object:

```
8450   CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
8451   CK_KEY_TYPE keyType = CKK_DSA;  
8452   CK_UTF8CHAR label[] = "A DSA public key object";  
8453   CK_BYTE prime[] = {...};  
8454   CK_BYTE subprime[] = {...};  
8455   CK_BYTE base[] = {...};
```

```
8456 CK_BYTE value[] = {...};
8457 CK_BBOOL true = CK_TRUE;
8458 CK_ATTRIBUTE template[] = {
8459     {CKA_CLASS, &class, sizeof(class)},
8460     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8461     {CKA_TOKEN, &true, sizeof(true)},
8462     {CKA_LABEL, label, sizeof(label)-1},
8463     {CKA_PRIME, prime, sizeof(prime)},
8464     {CKA_SUBPRIME, subprime, sizeof(subprime)},
8465     {CKA_BASE, base, sizeof(base)},
8466     {CKA_VALUE, value, sizeof(value)}
8467 };
8468
```

8469 **6.2.3 DSA Key Restrictions**

8470 [FIPS PUB 186-4] specifies permitted combinations of prime and sub-prime lengths. They are:

- 8471 • Prime: 1024 bits, Subprime: 160
- 8472 • Prime: 2048 bits, Subprime: 224
- 8473 • Prime: 2048 bits, Subprime: 256
- 8474 • Prime: 3072 bits, Subprime: 256

8475 Earlier versions of FIPS PUB 186 permitted smaller prime lengths, and those are included here for
8476 backwards compatibility. An implementation that is compliant to [FIPS PUB 186-4] does not permit the
8477 use of primes of any length less than 1024 bits.

8478 **6.2.4 DSA private key objects**

8479 DSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_DSA**) hold DSA private keys.
8480 The following table defines the DSA private key object attributes, in addition to the common attributes
8481 defined for this object class:

8482 *Table 57, DSA Private Key Object Attributes*

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

8483 ¹Refer to Table 13 for footnotes

8484 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
8485 parameters”. See [FIPS PUB 186-4] for more information on DSA keys.

8486 Note that when generating a DSA private key, the DSA domain parameters are *not* specified in the key’s
8487 template. This is because DSA private keys are only generated as part of a DSA key *pair*, and the DSA
8488 domain parameters for the pair are specified in the template for the DSA public key.

8489 The following is a sample template for creating a DSA private key object:

```
8490 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
8491 CK_KEY_TYPE keyType = CKK_DSA;
8492 CK_UTF8CHAR label[] = "A DSA private key object";
8493 CK_BYTE subject[] = {...};
8494 CK_BYTE id[] = {123};
```

```
8495 CK_BYTE prime[] = {...};
8496 CK_BYTE subprime[] = {...};
8497 CK_BYTE base[] = {...};
8498 CK_BYTE value[] = {...};
8499 CK_BBOOL true = CK_TRUE;
8500 CK_ATTRIBUTE template[] = {
8501     {CKA_CLASS, &class, sizeof(class)},
8502     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8503     {CKA_TOKEN, &true, sizeof(true)},
8504     {CKA_LABEL, label, sizeof(label)-1},
8505     {CKA_SUBJECT, subject, sizeof(subject)},
8506     {CKA_ID, id, sizeof(id)},
8507     {CKA_SENSITIVE, &true, sizeof(true)},
8508     {CKA_SIGN, &true, sizeof(true)},
8509     {CKA_PRIME, prime, sizeof(prime)},
8510     {CKA_SUBPRIME, subprime, sizeof(subprime)},
8511     {CKA_BASE, base, sizeof(base)},
8512     {CKA_VALUE, value, sizeof(value)}
8513 };
```

8514 **6.2.5 DSA domain parameter objects**

8515 DSA domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type **CKK_DSA**) hold
8516 DSA domain parameters. The following table defines the DSA domain parameter object attributes, in
8517 addition to the common attributes defined for this object class:

8518 *Table 58, 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.

8519 ¹ Refer to Table 13 for footnotes

8520 The **CKA_PRIME**, **CKA_SUBPRIME** and **CKA_BASE** attribute values are collectively the “DSA domain
8521 parameters”. See [FIPS PUB 186-4] for more information on DSA domain parameters.

8522 To ensure backwards compatibility, if **CKA_SUBPRIME_BITS** is not specified for a call to
8523 **C_GenerateKey**, it takes on a default based on the value of **CKA_PRIME_BITS** as follows:

- 8524 • If **CKA_PRIME_BITS** is less than or equal to 1024 then **CKA_SUBPRIME_BITS** shall be 160 bits
- 8525 • If **CKA_PRIME_BITS** equals 2048 then **CKA_SUBPRIME_BITS** shall be 224 bits
- 8526 • If **CKA_PRIME_BITS** equals 3072 then **CKA_SUBPRIME_BITS** shall be 256 bits

8527

8528 The following is a sample template for creating a DSA domain parameter object:

```
8529 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
8530 CK_KEY_TYPE keyType = CKK_DSA;
8531 CK_UTF8CHAR label[] = "A DSA domain parameter object";
8532 CK_BYTE prime[] = {...};
8533 CK_BYTE subprime[] = {...};
8534 CK_BYTE base[] = {...};
```

```

8535     CK_BBOOL true = CK_TRUE;
8536     CK_ATTRIBUTE template[] = {
8537         {CKA_CLASS, &class, sizeof(class)},
8538         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
8539         {CKA_TOKEN, &true, sizeof(true)},
8540         {CKA_LABEL, label, sizeof(label)-1},
8541         {CKA_PRIME, prime, sizeof(prime)},
8542         {CKA_SUBPRIME, subprime, sizeof(subprime)},
8543         {CKA_BASE, base, sizeof(base)},
8544     };

```

8545 6.2.6 DSA key pair generation

8546 The DSA key pair generation mechanism, denoted **CKM_DSA_KEY_PAIR_GEN**, is a key pair generation
8547 mechanism based on the Digital Signature Algorithm defined in FIPS PUB 186-2.

8548 This mechanism does not have a parameter.

8549 The mechanism generates DSA public/private key pairs with a particular prime, subprime and base, as
8550 specified in the **CKA_PRIME**, **CKA_SUBPRIME**, and **CKA_BASE** attributes of the template for the public
8551 key.

8552 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
8553 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**, **CKA_BASE**, and
8554 **CKA_VALUE** attributes to the new private key. Other attributes supported by the DSA public and private
8555 key types (specifically, the flags indicating which functions the keys support) may also be specified in the
8556 templates for the keys, or else are assigned default initial values.

8557 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8558 specify the supported range of DSA prime sizes, in bits.

8559 6.2.7 DSA domain parameter generation

8560 The DSA domain parameter generation mechanism, denoted **CKM_DSA_PARAMETER_GEN**, is a
8561 domain parameter generation mechanism based on the Digital Signature Algorithm defined in FIPS PUB
8562 186-2.

8563 This mechanism does not have a parameter.

8564 The mechanism generates DSA domain parameters with a particular prime length in bits, as specified in
8565 the **CKA_PRIME_BITS** attribute of the template.

8566 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
8567 **CKA_BASE** and **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the DSA
8568 domain parameter types may also be specified in the template, or else are assigned default initial values.

8569 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8570 specify the supported range of DSA prime sizes, in bits.

8571 6.2.8 DSA probabilistic domain parameter generation

8572 The DSA probabilistic domain parameter generation mechanism, denoted
8573 **CKM_DSA_PROBABILISTIC_PARAMETER_GEN**, is a domain parameter generation mechanism based
8574 on the Digital Signature Algorithm defined in [FIPS PUB 186-4], section Appendix A.1.1 Generation and
8575 Validation of Probable Primes..

8576 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
8577 returns the seed (*pSeed*) and the length (*ulSeedLen*).

8578 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
8579 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
8580 specified in the **CKA_SUBPRIME_BITS** attribute of the template.

8581 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
8582 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
8583 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
8584 template, or else are assigned default initial values.

8585 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8586 specify the supported range of DSA prime sizes, in bits.

8587 6.2.9 DSA Shawe-Taylor domain parameter generation

8588 The DSA Shawe-Taylor domain parameter generation mechanism, denoted
8589 **CKM_DSA_SHAWE_TAYLOR_PARAMETER_GEN**, is a domain parameter generation mechanism
8590 based on the Digital Signature Algorithm defined in [FIPS PUB 186-4], section Appendix A.1.2
8591 Construction and Validation of Provable Primes p and q.

8592 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash and
8593 returns the seed (pSeed) and the length (ulSeedLen).

8594 The mechanism generates DSA the prime and subprime domain parameters with a particular prime
8595 length in bits, as specified in the **CKA_PRIME_BITS** attribute of the template and the subprime length as
8596 specified in the **CKA_SUBPRIME_BITS** attribute of the template.

8597 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_SUBPRIME**,
8598 **CKA_PRIME_BITS**, and **CKA_SUBPRIME_BITS** attributes to the new object. **CKA_BASE** is not set by
8599 this call. Other attributes supported by the DSA domain parameter types may also be specified in the
8600 template, or else are assigned default initial values.

8601 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8602 specify the supported range of DSA prime sizes, in bits.

8603 6.2.10 DSA base domain parameter generation

8604 The DSA base domain parameter generation mechanism, denoted **CKM_DSA_FIPS_G_GEN**, is a base
8605 parameter generation mechanism based on the Digital Signature Algorithm defined in [FIPS PUB 186-4],
8606 section Appendix A.2 Generation of Generator G.

8607 This mechanism takes a **CK_DSA_PARAMETER_GEN_PARAM** which supplies the base hash the seed
8608 (pSeed) and the length (ulSeedLen) and the index value.

8609 The mechanism generates the DSA base with the domain parameter specified in the **CKA_PRIME** and
8610 **CKA_SUBPRIME** attributes of the template.

8611 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_BASE** attributes to the new
8612 object. Other attributes supported by the DSA domain parameter types may also be specified in the
8613 template, or else are assigned default initial values.

8614 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
8615 specify the supported range of DSA prime sizes, in bits.

8616 6.2.11 DSA without hashing

8617 The DSA without hashing mechanism, denoted **CKM_DSA**, is a mechanism for single-part signatures and
8618 verification based on the Digital Signature Algorithm defined in FIPS PUB 186-2. (This mechanism
8619 corresponds only to the part of DSA that processes the 20-byte hash value; it does not compute the hash
8620 value.)

8621 For the purposes of this mechanism, a DSA signature is a 40-byte string, corresponding to the
8622 concatenation of the DSA values *r* and *s*, each represented most-significant byte first.

8623 It does not have a parameter.

8624 Constraints on key types and the length of data are summarized in the following table:

Table 59, 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 hashing

The DSA with hashing mechanism, denoted **CKM_DSA_<hash>** where <hash> identifies a hash function as per Table 60, 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.

Table 60, DSA with hashing: mechanisms and hash functions

Mechanism	Hash function
CKM_DSA_SHA1	SHA-1
CKM_DSA_SHA224	SHA-224
CKM_DSA_SHA256	SHA-256
CKM_DSA_SHA384	SHA-384
CKM_DSA_SHA512	SHA-512
CKM_DSA_SHA3_224	SHA3-224
CKM_DSA_SHA3_256	SHA3-256
CKM_DSA_SHA3_384	SHA3-384
CKM_DSA_SHA3_512	SHA3-512

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 61, DSA with SHA-1: Key And Data Length

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

8649 L = 2048, N = 224
 8650 L = 2048, N = 256
 8651 L = 3072, N = 256
 8652

8653 6.3 Elliptic Curve

8654 The Elliptic Curve (EC) cryptosystem in this document was originally based on the one described in the
 8655 [ANSI X9.62] and [ANSI X9.63] standards developed by the ANSI X9F1 working group.

8656 The EC cryptosystem developed by the ANSI X9F1 working group was created at a time when EC curves
 8657 were always represented in their Weierstrass form. Since that time, new curves represented in Edwards
 8658 form ([RFC 8032]) and Montgomery form ([RFC 7748]) have become more common. To support these
 8659 new curves, the EC cryptosystem in this document has been extended from the original. Additional key
 8660 generation mechanisms have been added as well as an additional signature generation mechanism.

8661

8662 Table 62, Elliptic Curve Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_EC_KEY_PAIR_GEN					✓			
CKM_EC_KEY_PAIR_GEN_W_E XTRA_BITS					✓			
CKM_EC_EDWARDS_KEY_PAIR _GEN					✓			
CKM_EC_MONTGOMERY_KEY_ PAIR_GEN					✓			
CKM_ECDSA		✓ ¹						
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_DERI VE							✓	✓
CKM_ECMQV_DERIVE							✓	
CKM_ECDH_AES_KEY_WRAP						✓		
CKM_ECDH_COF_AES_KEY_WR AP						✓		

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ECDH_X_AES_KEY_WRAP						✓		

1 Single-part operations only

Table 63, 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

8686 supporting the compressed form. A Cryptoki library that can perform EC mechanisms must set either or
8687 both of these flags for each EC mechanism.

8688 Note that an implementation of a Cryptoki library supporting EC with only one variety, one representation
8689 of domain parameters or one form may encounter difficulties achieving interoperability with other
8690 implementations.

8691 If an attempt to create, generate, derive or unwrap an EC key of an unsupported curve is made, the
8692 attempt should fail with the error code **CKR_CURVE_NOT_SUPPORTED**. If an attempt to create,
8693 generate, derive, or unwrap an EC key with invalid or of an unsupported representation of domain
8694 parameters is made, that attempt should fail with the error code **CKR_DOMAIN_PARAMS_INVALID**. If
8695 an attempt to create, generate, derive, or unwrap an EC key of an unsupported form is made, that
8696 attempt should fail with the error code **CKR_TEMPLATE_INCONSISTENT**.

8697 6.3.1 EC Signatures

8698 For the purposes of these mechanisms, an ECDSA signature is an octet string of even length which is at
8699 most two times $nLen$ octets, where $nLen$ is the length in octets of the base point order n . The signature
8700 octets correspond to the concatenation of the ECDSA values r and s , both represented as an octet string
8701 of equal length of at most $nLen$ with the most significant byte first. If r and s have different octet length,
8702 the shorter of both must be padded with leading zero octets such that both have the same octet length.
8703 Loosely spoken, the first half of the signature is r and the second half is s . For signatures created by a
8704 token, the resulting signature is always of length $2nLen$. For signatures passed to a token for verification,
8705 the signature may have a shorter length but must be composed as specified before.

8706 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
8707 length of n will be used. Any truncation is done by the token.

8708 Note: For applications, it is recommended to encode the signature as an octet string of length two times
8709 $nLen$ if possible. This ensures that the application works with PKCS #11 modules which have been
8710 implemented based on an older version of this document. Older versions required all signatures to have
8711 length two times $nLen$. It may be impossible to encode the signature with the maximum length of two
8712 times $nLen$ if the application just gets the integer values of r and s (i.e. without leading zeros), but does
8713 not know the base point order n , because r and s can have any value between zero and the base point
8714 order n .

8715 An EdDSA signature is an octet string of even length which is two times $nLen$ octets, where $nLen$ is
8716 calculated as EdDSA parameter b divided by 8. The signature octets correspond to the concatenation of
8717 the EdDSA values R and S as defined in [RFC 8032], both represented as an octet string of equal length
8718 of $nLen$ bytes in little endian order.

8719 6.3.2 Definitions

8720 This section defines the key types "**CKK_EC**", "**CKK_EC_EDWARDS**" and "**CKK_EC_MONTGOMERY**"
8721 for type CK_KEY_TYPE as used in the **CKA_KEY_TYPE** attribute of key objects.

8722 Note: **CKK_ECDSA** is deprecated. It is replaced by **CKK_EC**.

8723 Mechanisms:

8724

8725 CKM_EC_KEY_PAIR_GEN

8726 CKM_EC_EDWARDS_KEY_PAIR_GEN

8727 CKM_EC_MONTGOMERY_KEY_PAIR_GEN

8728 CKM_ECDSA

8729 CKM_ECDSA_SHA1

8730 CKM_ECDSA_SHA224

8731 CKM_ECDSA_SHA256

8732 CKM_ECDSA_SHA384

8733	CKM_ECDSA_SHA512
8734	CKM_ECDSA_SHA3_224
8735	CKM_ECDSA_SHA3_256
8736	CKM_ECDSA_SHA3_384
8737	CKM_ECDSA_SHA3_512
8738	CKM_EDDSA
8739	CKM_XEDDSA
8740	CKM_ECDH1_DERIVE
8741	CKM_ECDH1_COFACTOR_DERIVE
8742	CKM_ECMQV_DERIVE
8743	CKM_ECDH_AES_KEY_WRAP
8744	CKM_ECDH_COF_AES_KEY_WRAP
8745	CKM_ECDH_X_AES_KEY_WRAP
8746	
8747	CKD_NULL
8748	CKD_SHA1_KDF
8749	CKD_SHA224_KDF
8750	CKD_SHA256_KDF
8751	CKD_SHA384_KDF
8752	CKD_SHA512_KDF
8753	CKD_SHA3_224_KDF
8754	CKD_SHA3_256_KDF
8755	CKD_SHA3_384_KDF
8756	CKD_SHA3_512_KDF
8757	CKD_SHA1_KDF_SP800
8758	CKD_SHA224_KDF_SP800
8759	CKD_SHA256_KDF_SP800
8760	CKD_SHA384_KDF_SP800
8761	CKD_SHA512_KDF_SP800
8762	CKD_SHA3_224_KDF_SP800
8763	CKD_SHA3_256_KDF_SP800
8764	CKD_SHA3_384_KDF_SP800
8765	CKD_SHA3_512_KDF_SP800
8766	CKD_BLAKE2B_160_KDF
8767	CKD_BLAKE2B_256_KDF
8768	CKD_BLAKE2B_384_KDF
8769	CKD_BLAKE2B_512_KDF

8770 6.3.3 Short Weierstrass Elliptic Curve public key objects

8771 Short Weierstrass EC public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_EC**) hold EC
8772 public keys. The following table defines the EC public key object attributes, in addition to the common
8773 attributes defined for this object class:

8774 Table 64, 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

8775 Refer to Table 13 for footnotes

8776 Note: **CKA_ECDSA_PARAMS** is deprecated. It is replaced by **CKA_EC_PARAMS**.

8777 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in [ANSI
8778 X9.62] as a choice of three parameter representation methods with the following syntax:

```
8779     Parameters ::= CHOICE {  
8780         ecParameters      ECPParameters,  
8781         oId                CURVES.&id({CurveNames}),  
8782         implicitlyCA       NULL,  
8783         curveName          PrintableString  
8784     }
```

8785
8786 This allows detailed specification of all required values using choice **ecParameters**, the use of **old** as an
8787 object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to
8788 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve
8789 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **old** or
8790 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used
8791 in Cryptoki.

8792 The following is a sample template for creating an short Weierstrass EC public key object:

```
8793     CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
8794     CK_KEY_TYPE keyType = CKK_EC;  
8795     CK_UTF8CHAR label[] = "An EC public key object";  
8796     CK_BYTE ecParams[] = {...};  
8797     CK_BYTE ecPoint[] = {...};  
8798     CK_BBOOL true = CK_TRUE;  
8799     CK_ATTRIBUTE template[] = {  
8800         {CKA_CLASS, &class, sizeof(class)},  
8801         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
8802         {CKA_TOKEN, &true, sizeof(true)},  
8803         {CKA_LABEL, label, sizeof(label)-1},  
8804         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},  
8805         {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}  
8806     };
```

8807 6.3.4 Short Weierstrass Elliptic Curve private key objects

8808 Short Weierstrass EC private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_EC**) hold
8809 EC private keys. See Section 6.3 for more information about EC. The following table defines the EC
8810 private key object attributes, in addition to the common attributes defined for this object class:

8811 Table 65, 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>

8812 Refer to Table 13 for footnotes

8813 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in [ANSI
8814 X9.62] as a choice of three parameter representation methods with the following syntax:

```
8815     Parameters ::= CHOICE {  
8816         ecParameters      ECPParameters,  
8817         oId                CURVES.&id({CurveNames}),  
8818         implicitlyCA       NULL,  
8819         curveName          PrintableString  
8820     }  
8821
```

8822 This allows detailed specification of all required values using choice **ecParameters**, the use of **old** as an
8823 object identifier substitute for a particular set of Elliptic Curve domain parameters, or **implicitlyCA** to
8824 indicate that the domain parameters are explicitly defined elsewhere, or **curveName** to specify a curve
8825 name as e.g. define in [ANSI X9.62], [BRAINPOOL], [SEC 2], [LEGIFRANCE]. The use of **old** or
8826 **curveName** is recommended over the choice **ecParameters**. The choice **implicitlyCA** must not be used
8827 in Cryptoki. Note that when generating an EC private key, the EC domain parameters are *not* specified in
8828 the key’s template. This is because EC private keys are only generated as part of an EC key *pair*, and the
8829 EC domain parameters for the pair are specified in the template for the EC public key.

8830 The following is a sample template for creating an short Weierstrass EC private key object:

```
8831     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
8832     CK_KEY_TYPE keyType = CKK_EC;  
8833     CK_UTF8CHAR label[] = “An EC private key object”;  
8834     CK_BYTE subject[] = {...};  
8835     CK_BYTE id[] = {123};  
8836     CK_BYTE ecParams[] = {...};  
8837     CK_BYTE value[] = {...};  
8838     CK_BBOOL true = CK_TRUE;  
8839     CK_ATTRIBUTE template[] = {  
8840         {CKA_CLASS, &class, sizeof(class)},  
8841         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
8842         {CKA_TOKEN, &true, sizeof(true)},  
8843         {CKA_LABEL, label, sizeof(label)-1},  
8844         {CKA_SUBJECT, subject, sizeof(subject)},  
8845         {CKA_ID, id, sizeof(id)},  
8846         {CKA_SENSITIVE, &true, sizeof(true)},  
8847         {CKA_DERIVE, &true, sizeof(true)},  
8848         {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},  
8849         {CKA_VALUE, value, sizeof(value)}  
8850     };
```

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 66, 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]

¹ Refer to Table 13 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:

8891 Table 67, 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]

8892 Refer to Table 13 for footnotes

8893 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in [ANSI
8894 X9.62] as a choice of three parameter representation methods. A 4th choice is added to support Edwards
8895 and Montgomery Elliptic Curves. The **CKA_EC_PARAMS** attribute has the following syntax:

```
8896 Parameters ::= CHOICE {  
8897     ecParameters      ECPParameters,  
8898     oId                CURVES.&id({CurveNames}),  
8899     implicitlyCA       NULL,  
8900     curveName          PrintableString  
8901 }
```

8902 Edwards EC private keys only support the use of the **curveName** selection to specify a curve name as
8903 defined in [RFC 8032] and the use of the **oId** selection to specify a curve through an EdDSA algorithm as
8904 defined in [RFC 8410]. Note that keys defined by [RFC 8032] and [RFC 8410] are incompatible.

8905 Note that when generating an Edwards EC private key, the EC domain parameters are *not* specified in
8906 the key’s template. This is because Edwards EC private keys are only generated as part of an Edwards
8907 EC key *pair*, and the EC domain parameters for the pair are specified in the template for the Edwards EC
8908 public key.

8909 The following is a sample template for creating an Edwards EC private key object:

```
8910 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
8911 CK_KEY_TYPE keyType = CKK_EC_EDWARDS;  
8912 CK_UTF8CHAR label[] = "An Edwards EC private key object";  
8913 CK_BYTE subject[] = {...};  
8914 CK_BYTE id[] = {123};  
8915 CK_BYTE ecParams[] = {...};  
8916 CK_BYTE value[] = {...};  
8917 CK_BBOOL true = CK_TRUE;  
8918 CK_ATTRIBUTE template[] = {  
8919     {CKA_CLASS, &class, sizeof(class)},  
8920     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
8921     {CKA_TOKEN, &true, sizeof(true)},  
8922     {CKA_LABEL, label, sizeof(label)-1},  
8923     {CKA_SUBJECT, subject, sizeof(subject)},  
8924     {CKA_ID, id, sizeof(id)},  
8925     {CKA_SENSITIVE, &true, sizeof(true)},  
8926     {CKA_DERIVE, &true, sizeof(true)},  
8927     {CKA_VALUE, value, sizeof(value)}  
8928 };
```

8929 6.3.7 Montgomery Elliptic Curve public key objects

8930 Montgomery EC public key objects (object class **CKO_PUBLIC_KEY**, key type
8931 **CKK_EC_MONTGOMERY**) hold Montgomery EC public keys. The following table defines the

8932 Montgomery EC public key object attributes, in addition to the common attributes defined for this object
8933 class:

8934 Table 68, 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]

8935 Refer to Table 13 for footnotes

8936 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
8937 [X9.62] as a choice of three parameter representation methods. A 4th choice is added to support Edwards
8938 and Montgomery Elliptic Curves. The **CKA_EC_PARAMS** attribute has the following syntax:

```
8939 Parameters ::= CHOICE {  
8940     ecParameters      ECPParameters,  
8941     oId                CURVES.&id({CurveNames}),  
8942     implicitlyCA       NULL,  
8943     curveName          PrintableString  
8944 }
```

8945 Montgomery EC public keys only support the use of the **curveName** selection to specify a curve name as
8946 defined in [RFC 7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm as
8947 defined in [RFC 8410]. Note that keys defined by [RFC 7748] and [RFC 8410] are incompatible.

8948 The following is a sample template for creating a Montgomery EC public key object:

```
8949 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
8950 CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;  
8951 CK_UTF8CHAR label[] = "A Montgomery EC public key object";  
8952 CK_BYTE ecParams[] = {...};  
8953 CK_BYTE ecPoint[] = {...};  
8954 CK_BBOOL true = CK_TRUE;  
8955 CK_ATTRIBUTE template[] = {  
8956     {CKA_CLASS, &class, sizeof(class)},  
8957     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
8958     {CKA_TOKEN, &true, sizeof(true)},  
8959     {CKA_LABEL, label, sizeof(label)-1},  
8960     {CKA_EC_PARAMS, ecParams, sizeof(ecParams)},  
8961     {CKA_EC_POINT, ecPoint, sizeof(ecPoint)}  
8962 };
```

8963 6.3.8 Montgomery Elliptic Curve private key objects

8964 Montgomery EC private key objects (object class **CKO_PRIVATE_KEY**, key type
8965 **CKK_EC_MONTGOMERY**) hold Montgomery EC private keys. See Section 6.3 for more information
8966 about EC. The following table defines the Montgomery EC private key object attributes, in addition to the
8967 common attributes defined for this object class:

8968 Table 69, 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]

8969 Refer to Table 13 for footnotes

8970 The **CKA_EC_PARAMS** attribute value is known as the “EC domain parameters” and is defined in ANSI
8971 [X9.62] as a choice of three parameter representation methods. A 4th choice is added to support Edwards
8972 and Montgomery Elliptic Curves. The **CKA_EC_PARAMS** attribute has the following syntax:

```
8973 Parameters ::= CHOICE {  
8974     ecParameters      ECPParameters,  
8975     oId                CURVES.&id({CurveNames}),  
8976     implicitlyCA       NULL,  
8977     curveName          PrintableString  
8978 }
```

8979 Montgomery EC private keys only support the use of the **curveName** selection to specify a curve name
8980 as defined in [RFC7748] and the use of the **oId** selection to specify a curve through an ECDH algorithm
8981 as defined in [RFC 8410]. Note that keys defined by [RFC 7748] and [RFC 8410] are incompatible.

8982 Note that when generating a Montgomery EC private key, the EC domain parameters are *not* specified in
8983 the key’s template. This is because Montgomery EC private keys are only generated as part of a
8984 Montgomery EC key *pair*, and the EC domain parameters for the pair are specified in the template for the
8985 Montgomery EC public key.

8986 The following is a sample template for creating a Montgomery EC private key object:

```
8987 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
8988 CK_KEY_TYPE keyType = CKK_EC_MONTGOMERY;  
8989 CK_UTF8CHAR label[] = "A Montgomery EC private key object";  
8990 CK_BYTE subject[] = {...};  
8991 CK_BYTE id[] = {123};  
8992 CK_BYTE ecParams[] = {...};  
8993 CK_BYTE value[] = {...};  
8994 CK_BBOOL true = CK_TRUE;  
8995 CK_ATTRIBUTE template[] = {  
8996     {CKA_CLASS, &class, sizeof(class)},  
8997     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
8998     {CKA_TOKEN, &true, sizeof(true)},  
8999     {CKA_LABEL, label, sizeof(label)-1},  
9000     {CKA_SUBJECT, subject, sizeof(subject)},  
9001     {CKA_ID, id, sizeof(id)},  
9002     {CKA_SENSITIVE, &true, sizeof(true)},  
9003     {CKA_DERIVE, &true, sizeof(true)},  
9004     {CKA_VALUE, value, sizeof(value)}  
9005     };
```

9006 **6.3.9 Elliptic Curve key pair generation**

9007 The short Weierstrass EKey pair generation mechanism, denoted **CKM_EC_KEY_PAIR_GEN**, is a key
9008 pair generation mechanism that uses the method defined by the [ANSI X9.62] and [ANSI X9.63]
9009 standards.

9010 The short Weierstrass EC key pair generation mechanism, denoted
9011 **CKM_EC_KEY_PAIR_GEN_W_EXTRA_BITS**, is a key pair generation mechanism that uses the method
9012 defined by [FIPS PUB 186-4] Appendix B.4.1.

9013 These mechanisms do not have a parameter.

9014 These mechanisms generate EC public/private key pairs with particular EC domain parameters, as
9015 specified in the **CKA_EC_PARAMS** attribute of the template for the public key. Note that this version of
9016 Cryptoki does not include a mechanism for generating these EC domain parameters.

9017 These mechanism contribute the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
9018 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
9019 attributes to the new private key. Other attributes supported by the EC public and private key types
9020 (specifically, the flags indicating which functions the keys support) may also be specified in the templates
9021 for the keys, or else are assigned default initial values.

9022 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9023 specify the minimum and maximum supported number of bits in the field sizes, respectively. For example,
9024 if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2^{200} and 2^{300}
9025 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the
9026 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
9027 301-bit number).

9028 6.3.10 Edwards Elliptic Curve key pair generation

9029 The Edwards EC key pair generation mechanism, denoted **CKM_EC_EDWARDS_KEY_PAIR_GEN**, is a
9030 key pair generation mechanism for EC keys over curves represented in Edwards form.

9031 This mechanism does not have a parameter.

9032 The mechanism can only generate EC public/private key pairs over the curves edwards25519 and
9033 edwards448 as defined in [RFC 8032] or the curves id-Ed25519 and id-Ed448 as defined in [RFC 8410].
9034 These curves can only be specified in the **CKA_EC_PARAMS** attribute of the template for the public key
9035 using the **curveName** or the old methods. Attempts to generate keys over these curves using any other
9036 EC key pair generation mechanism will fail with **CKR_CURVE_NOT_SUPPORTED**.

9037 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
9038 new public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_EC_PARAMS** and **CKA_VALUE**
9039 attributes to the new private key. Other attributes supported by the Edwards EC public and private key
9040 types (specifically, the flags indicating which functions the keys support) may also be specified in the
9041 templates for the keys, or else are assigned default initial values.

9042 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9043 specify the minimum and maximum supported number of bits in the field sizes, respectively. For this
9044 mechanism, the only allowed values are 255 and 448 as [RFC 8032] only defines curves of these two
9045 sizes. A Cryptoki implementation may support one or both of these curves and should set the
9046 *ulMinKeySize* and *ulMaxKeySize* fields accordingly.

9047 6.3.11 Montgomery Elliptic Curve key pair generation

9048 The Montgomery EC key pair generation mechanism, denoted
9049 **CKM_EC_MONTGOMERY_KEY_PAIR_GEN**, is a key pair generation mechanism for EC keys over
9050 curves represented in Montgomery form.

9051 This mechanism does not have a parameter.

9052 The mechanism can only generate Montgomery EC public/private key pairs over the curves curve25519
9053 and curve448 as defined in [RFC 7748] or the curves id-X25519 and id-X448 as defined in [RFC 8410].
9054 These curves can only be specified in the **CKA_EC_PARAMS** attribute of the template for the public key
9055 using the **curveName** or old methods. Attempts to generate keys over these curves using any other EC
9056 key pair generation mechanism will fail with **CKR_CURVE_NOT_SUPPORTED**.

9057 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_EC_POINT** attributes to the
9058 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 to 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 70, 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 hashing mechanism, denoted **CKM_ECDSA_<hash>** where <hash> identifies a hash function as per Table 71, is a mechanism for single- and multiple-part signatures and verification for ECDSA. This mechanism computes the entire ECDSA specification, including the hashing.

9090 Table 71, ECDSA with hashing: mechanisms and hash functions

Mechanism	Hash function
CKM_ECDSA_SHA1	SHA-1
CKM_ECDSA_SHA224	SHA-224
CKM_ECDSA_SHA256	SHA-256
CKM_ECDSA_SHA384	SHA-384
CKM_ECDSA_SHA512	SHA-512
CKM_ECDSA_SHA3_224	SHA3-224
CKM_ECDSA_SHA3_256	SHA3-256
CKM_ECDSA_SHA3_384	SHA3-384
CKM_ECDSA_SHA3_512	SHA3-512

9091

9092 This mechanism does not have a parameter.

9093 Constraints on key types and the length of data are summarized in the following table:

9094 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, $\leq 2nLen^2$	N/A

9095 2 Data length, signature length.

9096 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9097 specify the minimum and maximum supported number of bits in the field sizes, respectively. For example,
9098 if a Cryptoki library supports only ECDSA using a field of characteristic 2 which has between 2^{200} and 2^{300}
9099 elements, then *ulMinKeySize* = 201 and *ulMaxKeySize* = 301 (when written in binary notation, the
9100 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
9101 301-bit number).

9102 **6.3.14 EdDSA**

9103 The EdDSA mechanism, denoted **CKM_EDDSA**, is a mechanism for single-part and multipart signatures
9104 and verification for EdDSA. This mechanism implements the five EdDSA signature schemes defined in
9105 [RFC 8032] and [RFC 8410].

9106 For curves according to [RFC 8032], this mechanism has an optional parameter, a
9107 **CK_EDDSA_PARAMS** structure. The absence or presence of the parameter as well as its content is
9108 used to identify which signature scheme is to be used. The following table enumerates the five signature
9109 schemes defined in [RFC 8032] and all supported permutations of the mechanism parameter and its
9110 content.

9111 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

9112 For curves according to [RFC 8410], the mechanism is implicitly given by the curve, which is EdDSA in
9113 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	2b
C_Verify ¹	CKK_EC_MONTGOMERY public key	any, $\leq 2b$ 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 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;
```

9151 The fields of the structure have the following meanings:

9152	phFlag	a Boolean value which indicates if Pre-hashed variant of EdDSA should used
9153		
9154	ulContextDataLen	the length in bytes of the context data where $0 \leq \text{ulContextDataLen} \leq 255$.
9155		
9156	pContextData	context data shared between the signer and verifier

9157 **CK_EDDSA_PARAMS_PTR** is a pointer to a **CK_EDDSA_PARAMS**.

9158

9159 ♦ **CK_XEDDSA_PARAMS, CK_XEDDSA_PARAMS_PTR**

9160 **CK_XEDDSA_PARAMS** is a structure that provides the parameters for the **CKM_XEDDSA** signature mechanism. The structure is defined as follows:

```
9162 typedef struct CK_XEDDSA_PARAMS {
9163     CK_XEDDSA_HASH_TYPE hash;
9164 } CK_XEDDSA_PARAMS;
```

9165

9166 The fields of the structure have the following meanings:

9167	hash	a Hash mechanism to be used by the mechanism.
------	------	---

9168 **CK_XEDDSA_PARAMS_PTR** is a pointer to a **CK_XEDDSA_PARAMS**.

9169

9170 ♦ **CK_XEDDSA_HASH_TYPE, CK_XEDDSA_HASH_TYPE_PTR**

9171 **CK_XEDDSA_HASH_TYPE** is used to indicate the hash function used in XEDDSA. It is defined as follows:

```
9173 typedef CK_ULONG CK_XEDDSA_HASH_TYPE;
```

9174

9175 The following table lists the defined functions.

9176 *Table 76, EC: Key Derivation Functions*

Source Identifier
CKM_BLAKE2B_256
CKM_BLAKE2B_512
CKM_SHA3_256
CKM_SHA3_512
CKM_SHA256
CKM_SHA512

9177

9178 **CK_XEDDSA_HASH_TYPE_PTR** is a pointer to a **CK_XEDDSA_HASH_TYPE**.

9179

9180 ♦ **CK_EC_KDF_TYPE, CK_EC_KDF_TYPE_PTR**

9181 **CK_EC_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 EC key agreement schemes. It is defined as follows:

```
9184 typedef CK_ULONG CK_EC_KDF_TYPE;
```

9185

9186 The following table lists the defined functions.

9187 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

9188 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
9189 derivation function.

9190 The key derivation functions
9191 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**, which are
9192 based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
9193 respectively, derive keying data from the shared secret value as defined in [ANSI X9.63].

9194 The key derivation functions
9195 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**,
9196 which are based on SHA-1, SHA-224, SHA-384, SHA-512, SHA3-224, SHA3-256, SHA3-384, SHA3-512
9197 respectively, derive keying data from the shared secret value as defined in [NIST SP800-56A] section
9198 5.8.1.1.

9199 The key derivation functions **CKD_BLAKE2B_[160|256|384|512]_KDF**, which are based on the Blake2b
9200 family of hashes, derive keying data from the shared secret value as defined in [NIST SP800-56A] section
9201 5.8.1.1. **CK_EC_KDF_TYPE_PTR** is a pointer to a **CK_EC_KDF_TYPE**.

9202

9203 ♦ **CK_ECDH1_DERIVE_PARAMS, CK_ECDH1_DERIVE_PARAMS_PTR**

9204 **CK_ECDH1_DERIVE_PARAMS** is a structure that provides the parameters for the
9205 **CKM_ECDH1_DERIVE** and **CKM_ECDH1_COFACTOR_DERIVE** key derivation mechanisms, where
9206 each party contributes one key pair. The structure is defined as follows:

```
9207     typedef struct CK_ECDH1_DERIVE_PARAMS {  
9208         CK_EC_KDF_TYPE    kdf;  
9209         CK_ULONG           ulSharedDataLen;  
9210         CK_BYTE_PTR       pSharedData;  
9211         CK_ULONG           ulPublicDataLen;  
9212         CK_BYTE_PTR       pPublicData;  
9213     } CK_ECDH1_DERIVE_PARAMS;
```

9214

9215 The fields of the structure have the following meanings:

9216	kdf	key derivation function used on the shared secret value
9217	ulSharedDataLen	the length in bytes of the shared info
9218	pSharedData	some data shared between the two parties
9219	ulPublicDataLen	the length in bytes of the other party's EC public key
9220	pPublicData ¹	pointer to other party's EC public key value. For short Weierstrass
9221		EC keys: a token MUST be able to accept this value encoded as a
9222		raw octet string (as per section A.5.2 of [ANSI X9.62]). A token
9223		MAY, in addition, support accepting this value as a DER-encoded
9224		ECPoint (as per section E.6 of [ANSI X9.62]) i.e. the same as a
9225		CKA_EC_POINT encoding. The calling application is responsible for
9226		converting the offered public key to the compressed or
9227		uncompressed forms of these encodings if the token does not
9228		support the offered form.
9229		For Montgomery keys: the public key is provided as bytes in little
9230		endian order as defined in [RFC 7748].

9231 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be
9232 zero. With the key derivation functions

9233 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**,
9234 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**, an
9235 optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending
9236 to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

9237 **CK_ECDH1_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECDH1_DERIVE_PARAMS**.

9238 ♦ **CK_ECDH2_DERIVE_PARAMS, CK_ECDH2_DERIVE_PARAMS_PTR**

9239 **CK_ECDH2_DERIVE_PARAMS** is a structure that provides the parameters to the
9240 **CKM_ECMQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
9241 structure is defined as follows:

```
9242     typedef struct CK_ECDH2_DERIVE_PARAMS {  
9243         CK_EC_KDF_TYPE    kdf;  
9244         CK_ULONG           ulSharedDataLen;
```

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

```

9245     CK_BYTE_PTR pSharedData;
9246     CK_ULONG ulPublicDataLen;
9247     CK_BYTE_PTR pPublicData;
9248     CK_ULONG ulPrivateDataLen;
9249     CK_OBJECT_HANDLE hPrivateData;
9250     CK_ULONG ulPublicDataLen2;
9251     CK_BYTE_PTR pPublicData2;
9252 } CK_ECDH2_DERIVE_PARAMS;

```

9253

9254 The fields of the structure have the following meanings:

9255	kdf	key derivation function used on the shared secret value
9256	ulSharedDataLen	the length in bytes of the shared info
9257	pSharedData	some data shared between the two parties
9258	ulPublicDataLen	the length in bytes of the other party's first EC public key
9259	pPublicData	pointer to other party's first EC public key value. Encoding rules are
9260		as per pPublicData of CK_ECDH1_DERIVE_PARAMS
9261	ulPrivateDataLen	the length in bytes of the second EC private key
9262	hPrivateData	key handle for second EC private key value
9263	ulPublicDataLen2	the length in bytes of the other party's second EC public key
9264	pPublicData2	pointer to other party's second EC public key value. Encoding rules
9265		are as per pPublicData of CK_ECDH1_DERIVE_PARAMS

9266 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be
9267 zero. With the key derivation function **CKD_SHA1_KDF**, an optional *pSharedData* may be supplied,
9268 which consists of some data shared by the two parties intending to share the shared secret. Otherwise,
9269 *pSharedData* must be NULL and *ulSharedDataLen* must be zero.

9270 **CK_ECDH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECDH2_DERIVE_PARAMS**.

9271

9272 ♦ **CK_ECMQV_DERIVE_PARAMS, CK_ECMQV_DERIVE_PARAMS_PTR**

9273 **CK_ECMQV_DERIVE_PARAMS** is a structure that provides the parameters to the
9274 **CKM_ECMQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
9275 structure is defined as follows:

```

9276     typedef struct CK_ECMQV_DERIVE_PARAMS {
9277         CK_EC_KDF_TYPE      kdf;
9278         CK_ULONG             ulSharedDataLen;
9279         CK_BYTE_PTR         pSharedData;
9280         CK_ULONG             ulPublicDataLen;
9281         CK_BYTE_PTR         pPublicData;
9282         CK_ULONG             ulPrivateDataLen;
9283         CK_OBJECT_HANDLE     hPrivateData;
9284         CK_ULONG             ulPublicDataLen2;
9285         CK_BYTE_PTR         pPublicData2;
9286         CK_OBJECT_HANDLE     publicKey;
9287     } CK_ECMQV_DERIVE_PARAMS;

```

9288

9289	The fields of the structure have the following meanings:		
9290	kdf	key derivation function used on the shared secret value	
9291	ulSharedDataLen	the length in bytes of the shared info	
9292	pSharedData	some data shared between the two parties	
9293	ulPublicDataLen	the length in bytes of the other party's first EC public key	
9294	pPublicData	pointer to other party's first EC public key value. Encoding rules are	
9295		as per pPublicData of CK_ECDH1_DERIVE_PARAMS	
9296	ulPrivateDataLen	the length in bytes of the second EC private key	
9297	hPrivateData	key handle for second EC private key value	
9298	ulPublicDataLen2	the length in bytes of the other party's second EC public key	
9299	pPublicData2	pointer to other party's second EC public key value. Encoding rules	
9300		are as per pPublicData of CK_ECDH1_DERIVE_PARAMS	
9301	publicKey	Handle to the first party's ephemeral public key	

9302 With the key derivation function **CKD_NULL**, *pSharedData* must be NULL and *ulSharedDataLen* must be
9303 zero. With the key derivation functions
9304 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF**,
9305 **CKD_[SHA1|SHA224|SHA384|SHA512|SHA3_224|SHA3_256|SHA3_384|SHA3_512]_KDF_SP800**, an
9306 optional *pSharedData* may be supplied, which consists of some data shared by the two parties intending
9307 to share the shared secret. Otherwise, *pSharedData* must be NULL and *ulSharedDataLen* must be zero.
9308 **CK_ECMQV_DERIVE_PARAMS_PTR** is a pointer to a **CK_ECMQV_DERIVE_PARAMS**.

9309 6.3.17 Elliptic Curve Diffie-Hellman key derivation

9310 The Elliptic Curve Diffie-Hellman (ECDH) key derivation mechanism, denoted **CKM_ECDH1_DERIVE**, is
9311 a mechanism for key derivation based on the Diffie-Hellman version of the Elliptic Curve key agreement
9312 scheme, as defined in [ANSI X9.63] for short Weierstrass EC keys and [RFC 7748] for Montgomery keys,
9313 where each party contributes one key pair all using the same EC domain parameters.

9314 It has a parameter, a **CK_ECDH1_DERIVE_PARAMS** structure.

9315 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
9316 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
9317 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
9318 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
9319 type must be specified in the template.

9320 This mechanism has the following rules about key sensitivity and extractability:

- 9321 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
9322 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
9323 default value.
- 9324 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
9325 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
9326 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
9327 **CKA_SENSITIVE** attribute.
- 9328 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
9329 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
9330 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
9331 value from its **CKA_EXTRACTABLE** attribute.

9332 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9333 specify the minimum and maximum supported number of bits in the field sizes, respectively. For example,
9334 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).

When this mechanism is used in **C_EncapsulateKey** and **C_DecapsulateKey**, the mechanism parameters *pPublicData* and *ulPublicDataLen* must be set to NULL and 0 respectively. For **C_EncapsulateKey**, an ephemeral key pair is generated. The value of the generated public key is returned as the ciphertext. The generated private key is used with public key provided in the API to generate a symmetric key using EC Derive and has the same format as the public key used in **C_DeriveKey**. For **C_DecapsulateKey**, the ciphertext is used with the private key provided in the API to generate a symmetric key using EC Derive.

Constraints on key types are summarized in the following table:

Table 78, ECDH: Allowed Key Types

Function	Key type
C_DeriveKey	CKK_EC or CKK_EC_MONTGOMERY
C_EncapsulateKey	CKK_EC or CKK_EC_MONTGOMERY
C_DecapsulateKey	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. Cofactor multiplication is computationally efficient and helps to prevent security problems like small group attacks.

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

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

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

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

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

When this mechanism is used in **C_EncapsulateKey** and **C_DecapsulateKey**, the mechanism parameters *pPublicData* and *ulPublicDataLen* must be set to NULL and 0 respectively. For **C_EncapsulateKey**, an ephemeral key pair is generated. The value of the generated public key is

returned as the ciphertext. The generated private key is used with public key provided in the API to generate a symmetric key using EC Cofactor Derive and has the same format as the public key used in **C_DeriveKey**. For **C_DecapsulateKey**, the ciphertext is used with the private key provided in the API to generate a symmetric key using EC Cofactor Derive.

Constraints on key types are summarized in the following table:

Table 79, ECDH with cofactor: Allowed Key Types

Function	Key type
C_DeriveKey	CKK_EC
C_EncapsulateKey	CKK_EC
C_DecapsulateKey	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_DeriveKey	CKK_EC

6.3.20 ECDH AES KEY WRAP

Note: Mechanism **CKM_ECDH_AES_KEY_WRAP** is deprecated with PKCS #11 Version 3.2. Use **CKM_ECDH_COF_AES_KEY_WRAP** and **CKM_ECDH_X_AES_KEY_WRAP** instead.

9420 **CKM_ECDH_X_AES_KEY_WRAP** will work as **CKM_ECDH_AES_KEY_WRAP** applied to
 9421 **CKK_EC_MONTGOMERY** keys.
 9422

9423 The ECDH AES KEY WRAP mechanism, denoted **CKM_ECDH_AES_KEY_WRAP**, is a mechanism
 9424 based on Elliptic Curve public-key crypto-system and the AES key wrap mechanism. It supports single-
 9425 part key wrapping; and key unwrapping.

9426 It has a parameter, a **CK_ECDH_AES_KEY_WRAP_PARAMS** structure.
 9427

9428 The mechanism can wrap and unwrap an asymmetric target key of any length and type using an EC key.

- 9429 • A temporary AES key is derived from a temporary EC key and the wrapping EC key using the
 9430 **CKM_ECDH1_DERIVE** mechanism.
- 9431 • The derived AES key is used for wrapping the target key using the **CKM_AES_KEY_WRAP_KWP**
 9432 mechanism.

9433

9434 For wrapping, the mechanism -

- 9435 • Generates a temporary random EC key (transport key) having the same parameters as the
 9436 wrapping EC key (and domain parameters). Saves the transport key public key material.
- 9437 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen
 9438 and pSharedData using the private key of the transport EC key and the public key of wrapping EC
 9439 key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.
- 9440 • Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- 9441 • Zeroizes the temporary AES key and EC transport private key.
- 9442 • Concatenates public key material of the transport key and output the concatenated blob. The first
 9443 part is the public key material of the transport key and the second part is the wrapped target key.

9444

9445 The private target key will be encoded as defined in section 6.7.

9446

9447 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the object
 9448 attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.

9449

9450 For unwrapping, the mechanism -

- 9451 • Splits the input into two parts. The first part is the public key material of the transport key and the
 9452 second part is the wrapped target key. The length of the first part is equal to the length of the public
 9453 key material of the unwrapping EC key.

9454 *Note: since the transport key and the wrapping EC key share the same domain, the length of the*
 9455 *public key material of the transport key is the same length of the public key material of the*
 9456 *unwrapping EC key.*

- 9457 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen
 9458 and pSharedData using the private part of unwrapping EC key and the public part of the transport
 9459 EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.
- 9460 • Un-wraps the target key from the second part with the temporary AES key using
 9461 **CKM_AES_KEY_WRAP_KWP**.
- 9462 • Zeroizes the temporary AES key.

9463

9464 *Table 81, CKM_ECDH_AES_KEY_WRAP Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ECDH_AES_KEY_WRAP						✓		

Constraints on key types are summarized in the following table:

Table 82, ECDH AES Key Wrap: Allowed Key Types

Function	Key type
C_WrapKey / C_UnwrapKey	CKK_EC or CKK_EC_MONTGOMERY

6.3.21 ECDH COFACTOR AES KEY WRAP

The ECDH COFACTOR AES KEY WRAP mechanism, denoted **CKM_ECDH_COF_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 a **CKK_EC** key.

- A temporary AES key is derived from a temporary EC key and the wrapping EC key using the **CKM_ECDH1_COFACTOR_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 as DER-encoded OCTET STRING of the ANSI X9.62 ECPoint value.
- Performs ECDH operation using **CKM_ECDH1_COFACTOR_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. Since the public key material is a DER encoded OCTET STRING its length can be easily determined.
- Performs ECDH operation using **CKM_ECDH1_COFACTOR_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 83, CKM_ECDH_COF_AES_KEY_WRAP Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ECDH_COF_AES_KEY_WRAP						✓		

Constraints on key types are summarized in the following table:

Table 84, ECDH AES Key Wrap: Allowed Key Types

Function	Key type
C_WrapKey / C_UnwrapKey	CKK_EC

6.3.22 ECDH Montgomery AES KEY WRAP

The ECDH Montgomery AES KEY WRAP mechanism, denoted **CKM_ECDH_X_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 a **CKK_EC_MONTGOMERY** 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 -

- 9531 • Generates a temporary random EC key (transport key) having the same parameters as the
9532 wrapping **CKK_EC_MONTGOMERY** EC key.
- 9533 • Saves the transport key public key material as bytes in little endian order as defined in [RFC 7748]
9534 (i.e. as encoded in the **CKA_EC_POINT** attribute of Montgomery public keys).
- 9535 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen
9536 and pSharedData using the private key of the transport EC key and the public key of wrapping EC
9537 key and gets the first ulAESKeyBits bits of the derived key to be the temporary AES key.
- 9538 • Wraps the target key with the temporary AES key using **CKM_AES_KEY_WRAP_KWP**.
- 9539 • Zeroizes the temporary AES key and EC transport private key.
- 9540 • Concatenates public key material of the transport key and output the concatenated blob. The first
9541 part is the public key material of the transport key and the second part is the wrapped target key.
- 9542
- 9543 The private target key will be encoded as defined in section 6.7.
- 9544
- 9545 The use of Attributes in the PrivateKeyInfo structure is OPTIONAL. In case of conflicts between the object
9546 attribute template, and Attributes in the PrivateKeyInfo structure, an error should be thrown.
- 9547
- 9548 For unwrapping, the mechanism -
- 9549 • Splits the input into two parts. The first part is the public key material of the
9550 **CKK_EC_MONTGOMERY** transport key and the second part is the wrapped target key. The length
9551 of the first part is equal to the length of the public key material of the unwrapping
9552 **CKK_EC_MONTGOMERY** EC key.
- 9553 *Note: since the transport key and the wrapping EC key share the same domain, the length of the*
9554 *public key material of the transport key is the same length as the public key material of the*
9555 *unwrapping EC key.*
- 9556 • Performs ECDH operation using **CKM_ECDH1_DERIVE** with parameters of kdf, ulSharedDataLen
9557 and pSharedData using the private part of unwrapping EC key and the public part of the transport
9558 EC key and gets first ulAESKeyBits bits of the derived key to be the temporary AES key.
- 9559 • Un-wraps the target key from the second part with the temporary AES key using
9560 **CKM_AES_KEY_WRAP_KWP**.
- 9561 • Zeroizes the temporary AES key.
- 9562

9563 Table 85, CKM_ECDH_X_AES_KEY_WRAP Mechanisms vs. Functions

	Functions							
Mechanism	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ECDH_X_AES_KEY_WRAP						✓		

9564
9565 Constraints on key types are summarized in the following table:

9566 Table 86, ECDH AES Key Wrap: Allowed Key Types

Function	Key type
C_WrapKey / C_UnwrapKey	CKK_EC_MONTGOMERY

9567

9568 **6.3.23 ECDH AES KEY WRAP mechanism parameters**

9569 * **CK_ECDH_AES_KEY_WRAP_PARAMS; CK_ECDH_AES_KEY_WRAP_PARAMS_PTR**

9570 **CK_ECDH_AES_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
9571 **CKM_ECDH_AES_KEY_WRAP, CKM_ECDH_COF_AES_KEY_WRAP** and
9572 **CKM_ECDH_X_AES_KEY_WRAP** mechanism. It is defined as follows:

```
9573
9574     typedef struct CK_ECDH_AES_KEY_WRAP_PARAMS {
9575         CK_ULONG          ulAESKeyBits;
9576         CK_EC_KDF_TYPE    kdf;
9577         CK_ULONG          ulSharedDataLen;
9578         CK_BYTE_PTR       pSharedData;
9579     } CK_ECDH_AES_KEY_WRAP_PARAMS;
```

9580

9581 The fields of the structure have the following meanings:

9582

9583 ulAESKeyBits length of the temporary AES key in bits. Can be only 128, 192 or
9584 256.

9585 kdf key derivation function used on the shared secret value to generate
9586 AES key.

9587 ulSharedDataLen the length in bytes of the shared info

9588 pSharedData Some data shared between the two parties

9589

9590 **CK_ECDH_AES_KEY_WRAP_PARAMS_PTR** is a pointer to a
9591 **CK_ECDH_AES_KEY_WRAP_PARAMS**.

9592

9593 **6.3.24 FIPS 186-4**

9594 When **CKM_ECDSA** is operated in FIPS mode, the curves SHALL either be NIST recommended curves
9595 (with a fixed set of domain parameters) or curves with domain parameters generated as specified by
9596 [ANSI X9.62]. The NIST recommended curves are:

9597 P-192, P-224, P-256, P-384, P-521

9598 K-163, B-163, K-233, B-233

9599 K-283, B-283, K-409, B-409

9600 K-571, B-571

9601 **6.4 Diffie-Hellman**

9602 Table 87, Diffie-Hellman Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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 88, 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 13 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;
```

9627 CK_KEY_TYPE keyType = CKK_DH;
9628 CK_UTF8CHAR label[] = "A Diffie-Hellman public key object";
9629 CK_BYTE prime[] = {...};
9630 CK_BYTE base[] = {...};
9631 CK_BYTE value[] = {...};
9632 CK_BBOOL true = CK_TRUE;
9633 CK_ATTRIBUTE template[] = {
9634 {CKA_CLASS, &class, sizeof(class)},
9635 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
9636 {CKA_TOKEN, &true, sizeof(true)},
9637 {CKA_LABEL, label, sizeof(label)-1},
9638 {CKA_PRIME, prime, sizeof(prime)},
9639 {CKA_BASE, base, sizeof(base)},
9640 {CKA_VALUE, value, sizeof(value)}
9641 };

9642 **6.4.3 X9.42 Diffie-Hellman public key objects**

9643 X9.42 Diffie-Hellman public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_X9_42_DH**)
9644 hold X9.42 Diffie-Hellman public keys. The following table defines the X9.42 Diffie-Hellman public key
9645 object attributes, in addition to the common attributes defined for this object class:

9646 *Table 89, 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

9647 Refer to Table 13 for footnotes

9648 The **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attribute values are collectively the "X9.42 Diffie-
9649 Hellman domain parameters". See the [ANSI X9.42] standard for more information on X9.42 Diffie-
9650 Hellman keys.

9651 The following is a sample template for creating a X9.42 Diffie-Hellman public key object:

9652 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;
9653 CK_KEY_TYPE keyType = CKK_X9_42_DH;
9654 CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman public key
9655 object";
9656 CK_BYTE prime[] = {...};
9657 CK_BYTE base[] = {...};
9658 CK_BYTE subprime[] = {...};
9659 CK_BYTE value[] = {...};
9660 CK_BBOOL true = CK_TRUE;
9661 CK_ATTRIBUTE template[] = {
9662 {CKA_CLASS, &class, sizeof(class)},
9663 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
9664 {CKA_TOKEN, &true, sizeof(true)},
9665 {CKA_LABEL, label, sizeof(label)-1},
9666 {CKA_PRIME, prime, sizeof(prime)},

```
9667         {CKA_BASE, base, sizeof(base)},
9668         {CKA_SUBPRIME, subprime, sizeof(subprime)},
9669         {CKA_VALUE, value, sizeof(value)}
9670     };
```

9671 **6.4.4 Diffie-Hellman private key objects**

9672 Diffie-Hellman private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_DH**) hold Diffie-
9673 Hellman private keys. The following table defines the Diffie-Hellman private key object attributes, in
9674 addition to the common attributes defined for this object class:

9675 *Table 90, Diffie-Hellman Private Key Object Attributes*

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

9676 ¹Refer to Table 13 for footnotes

9677 The **CKA_PRIME** and **CKA_BASE** attribute values are collectively the “Diffie-Hellman domain
9678 parameters”. Depending on the token, there may be limits on the length of the key components. See
9679 [PKCS #3] for more information on Diffie-Hellman keys.

9680 Note that when generating a Diffie-Hellman private key, the Diffie-Hellman parameters are *not* specified in
9681 the key’s template. This is because Diffie-Hellman private keys are only generated as part of a Diffie-
9682 Hellman key *pair*, and the Diffie-Hellman parameters for the pair are specified in the template for the
9683 Diffie-Hellman public key.

9684 The following is a sample template for creating a Diffie-Hellman private key object:

```
9685     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
9686     CK_KEY_TYPE keyType = CKK_DH;
9687     CK_UTF8CHAR label[] = "A Diffie-Hellman private key object";
9688     CK_BYTE subject[] = {...};
9689     CK_BYTE id[] = {123};
9690     CK_BYTE prime[] = {...};
9691     CK_BYTE base[] = {...};
9692     CK_BYTE value[] = {...};
9693     CK_BBOOL true = CK_TRUE;
9694     CK_ATTRIBUTE template[] = {
9695         {CKA_CLASS, &class, sizeof(class)},
9696         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
9697         {CKA_TOKEN, &>true, sizeof(true)},
9698         {CKA_LABEL, label, sizeof(label)-1},
9699         {CKA_SUBJECT, subject, sizeof(subject)},
9700         {CKA_ID, id, sizeof(id)},
9701         {CKA_SENSITIVE, &>true, sizeof(true)},
9702         {CKA_DERIVE, &>true, sizeof(true)},
9703         {CKA_PRIME, prime, sizeof(prime)},
9704         {CKA_BASE, base, sizeof(base)},
9705         {CKA_VALUE, value, sizeof(value)}
9706     };
```

6.4.5 X9.42 Diffie-Hellman private key objects

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

Table 91, 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

Refer to Table 13 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 key components. See the [ANSI X9.42] standard for more information on X9.42 Diffie-Hellman keys.

Note that when generating a X9.42 Diffie-Hellman private key, the X9.42 Diffie-Hellman domain parameters are *not* specified in the key’s template. This is because X9.42 Diffie-Hellman private keys are only generated as part of a X9.42 Diffie-Hellman key *pair*, and the X9.42 Diffie-Hellman domain parameters for the pair are specified in the template for the X9.42 Diffie-Hellman public key.

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

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_X9_42_DH;
CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
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_SUBJECT, subject, sizeof(subject)},
    {CKA_ID, id, sizeof(id)},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_DERIVE, &true, sizeof(true)},
    {CKA_PRIME, prime, sizeof(prime)},
    {CKA_BASE, base, sizeof(base)},
    {CKA_SUBPRIME, subprime, sizeof(subprime)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.4.6 Diffie-Hellman domain parameter objects

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

Table 92, 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.

Refer to Table 13 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 domain parameters.

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

```
CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
CK_KEY_TYPE keyType = CKK_DH;
CK_UTF8CHAR label[] = "A Diffie-Hellman domain parameters
    object";
CK_BYTE prime[] = {...};
CK_BYTE base[] = {...};
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)},
};
```

6.4.7 X9.42 Diffie-Hellman domain parameters objects

X9.42 Diffie-Hellman domain parameters objects (object class **CKO_DOMAIN_PARAMETERS**, key type **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 93, 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 13 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

9779 parameters components. See the [ANSI X9.42] standard for more information on X9.42 Diffie-Hellman
9780 domain parameters.

9781 The following is a sample template for creating a X9.42 Diffie-Hellman domain parameters object:

```
9782 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;  
9783 CK_KEY_TYPE keyType = CKK_X9_42_DH;  
9784 CK_UTF8CHAR label[] = "A X9.42 Diffie-Hellman domain  
9785     parameters object";  
9786 CK_BYTE prime[] = {...};  
9787 CK_BYTE base[] = {...};  
9788 CK_BYTE subprime[] = {...};  
9789 CK_BBOOL true = CK_TRUE;  
9790 CK_ATTRIBUTE template[] = {  
9791     {CKA_CLASS, &class, sizeof(class)},  
9792     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
9793     {CKA_TOKEN, &true, sizeof(true)},  
9794     {CKA_LABEL, label, sizeof(label)-1},  
9795     {CKA_PRIME, prime, sizeof(prime)},  
9796     {CKA_BASE, base, sizeof(base)},  
9797     {CKA_SUBPRIME, subprime, sizeof(subprime)},  
9798     };
```

9799 6.4.8 PKCS #3 Diffie-Hellman key pair generation

9800 The PKCS #3 Diffie-Hellman key pair generation mechanism, denoted
9801 **CKM_DH_PKCS_KEY_PAIR_GEN**, is a key pair generation mechanism based on Diffie-Hellman key
9802 agreement, as defined in [PKCS #3]. This is what PKCS #3 calls "phase I". It does not have a parameter.

9803 The mechanism generates Diffie-Hellman public/private key pairs with a particular prime and base, as
9804 specified in the **CKA_PRIME** and **CKA_BASE** attributes of the template for the public key. If the
9805 **CKA_VALUE_BITS** attribute of the private key is specified, the mechanism limits the length in bits of the
9806 private value, as described in [PKCS #3].

9807 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
9808 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and **CKA_VALUE** (and
9809 the **CKA_VALUE_BITS** attribute, if it is not already provided in the template) attributes to the new private
9810 key; other attributes required by the Diffie-Hellman public and private key types must be specified in the
9811 templates.

9812 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9813 specify the supported range of Diffie-Hellman prime sizes, in bits.

9814 6.4.9 PKCS #3 Diffie-Hellman domain parameter generation

9815 The PKCS #3 Diffie-Hellman domain parameter generation mechanism, denoted
9816 **CKM_DH_PKCS_PARAMETER_GEN**, is a domain parameter generation mechanism based on Diffie-
9817 Hellman key agreement, as defined in [PKCS #3].

9818 It does not have a parameter.

9819 The mechanism generates Diffie-Hellman domain parameters with a particular prime length in bits, as
9820 specified in the **CKA_PRIME_BITS** attribute of the template.

9821 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, and
9822 **CKA_PRIME_BITS** attributes to the new object. Other attributes supported by the Diffie-Hellman domain
9823 parameter types may also be specified in the template, or else are assigned default initial values.

9824 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9825 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.

When this mechanism is used in **C_EncapsulateKey** and **C_DecapsulateKey**, the mechanism parameters *pPublicData* and *ulPublicDataLen* must be set to **NULL** and 0 respectively. For **C_EncapsulateKey**, an ephemeral key pair is generated. The value of the generated public key is returned as the ciphertext. The generated private key is used with public key provided in the API to generate a symmetric key using Diffie Helman PKCS #3 Derive. For **C_DecapsulateKey**, the ciphertext is used with the private key provided in the API to generate a symmetric key using Diffie Helman PKCS #3 Derive.

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.

6.4.11 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.

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**.

9868 Table 94, X9.42 Diffie-Hellman Key Derivation Functions

Source Identifier
CKD_NULL
CKD_SHA1_KDF_ASN1
CKD_SHA1_KDF_CONCATENATE

9869 The key derivation function **CKD_NULL** produces a raw shared secret value without applying any key
9870 derivation function whereas the key derivation functions **CKD_SHA1_KDF_ASN1** and
9871 **CKD_SHA1_KDF_CONCATENATE**, which are both based on SHA-1, derive keying data from the
9872 shared secret value as defined in the [ANSI X9.42] standard.

9873 **CK_X9_42_DH_KDF_TYPE_PTR** is a pointer to a **CK_X9_42_DH_KDF_TYPE**.

9874 ♦ **CK_X9_42_DH1_DERIVE_PARAMS, CK_X9_42_DH1_DERIVE_PARAMS_PTR**

9875 **CK_X9_42_DH1_DERIVE_PARAMS** is a structure that provides the parameters to the
9876 **CKM_X9_42_DH_DERIVE** key derivation mechanism, where each party contributes one key pair. The
9877 structure is defined as follows:

```
9878     typedef struct CK_X9_42_DH1_DERIVE_PARAMS {  
9879         CK_X9_42_DH_KDF_TYPE    kdf;  
9880         CK_ULONG                 ulOtherInfoLen;  
9881         CK_BYTE_PTR              pOtherInfo;  
9882         CK_ULONG                 ulPublicDataLen;  
9883         CK_BYTE_PTR              pPublicData;  
9884     } CK_X9_42_DH1_DERIVE_PARAMS;
```

9885

9886 The fields of the structure have the following meanings:

9887	kdf	key derivation function used on the shared secret value
9888	ulOtherInfoLen	the length in bytes of the other info
9889	pOtherInfo	some data shared between the two parties
9890	ulPublicDataLen	the length in bytes of the other party's X9.42 Diffie-Hellman public
9891		key
9892	pPublicData	pointer to other party's X9.42 Diffie-Hellman public key value

9893 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
9894 With the key derivation function **CKD_SHA1_KDF_ASN1**, *pOtherInfo* must be supplied, which contains
9895 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
9896 the two parties intending to share the shared secret. With the key derivation function
9897 **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
9898 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL
9899 and *ulOtherInfoLen* must be zero.

9900 **CK_X9_42_DH1_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH1_DERIVE_PARAMS**.

9901 • **CK_X9_42_DH2_DERIVE_PARAMS, CK_X9_42_DH2_DERIVE_PARAMS_PTR**

9902 **CK_X9_42_DH2_DERIVE_PARAMS** is a structure that provides the parameters to the
9903 **CKM_X9_42_DH_HYBRID_DERIVE** and **CKM_X9_42_MQV_DERIVE** key derivation mechanisms,
9904 where each party contributes two key pairs. The structure is defined as follows:

```
9905     typedef struct CK_X9_42_DH2_DERIVE_PARAMS {  
9906         CK_X9_42_DH_KDF_TYPE    kdf;
```

```

9907         CK_ULONG          ulOtherInfoLen;
9908         CK_BYTE_PTR        pOtherInfo;
9909         CK_ULONG          ulPublicDataLen;
9910         CK_BYTE_PTR        pPublicData;
9911         CK_ULONG          ulPrivateDataLen;
9912         CK_OBJECT_HANDLE   hPrivateKey;
9913         CK_ULONG          ulPublicDataLen2;
9914         CK_BYTE_PTR        pPublicData2;
9915     } CK_X9_42_DH2_DERIVE_PARAMS;

```

9916

9917 The fields of the structure have the following meanings:

9918	kdf	key derivation function used on the shared secret value
9919	ulOtherInfoLen	the length in bytes of the other info
9920	pOtherInfo	some data shared between the two parties
9921	ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman
9922		public key
9923	pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
9924	ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
9925	hPrivateKey	key handle for second X9.42 Diffie-Hellman private key value
9926	ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman
9927		public key
9928	pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key
9929		value

9930 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
9931 With the key derivation function **CKD_SHA1_KDF_ASN1**, *pOtherInfo* must be supplied, which contains
9932 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
9933 the two parties intending to share the shared secret. With the key derivation function
9934 **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
9935 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL
9936 and *ulOtherInfoLen* must be zero.

9937 **CK_X9_42_DH2_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_DH2_DERIVE_PARAMS**.

9938 • **CK_X9_42_MQV_DERIVE_PARAMS, CK_X9_42_MQV_DERIVE_PARAMS_PTR**

9939 **CK_X9_42_MQV_DERIVE_PARAMS** is a structure that provides the parameters to the
9940 **CKM_X9_42_MQV_DERIVE** key derivation mechanism, where each party contributes two key pairs. The
9941 structure is defined as follows:

```

9942     typedef struct CK_X9_42_MQV_DERIVE_PARAMS {
9943         CK_X9_42_DH_KDF_TYPE kdf;
9944         CK_ULONG          ulOtherInfoLen;
9945         CK_BYTE_PTR        pOtherInfo;
9946         CK_ULONG          ulPublicDataLen;
9947         CK_BYTE_PTR        pPublicData;
9948         CK_ULONG          ulPrivateDataLen;
9949         CK_OBJECT_HANDLE   hPrivateKey;
9950         CK_ULONG          ulPublicDataLen2;
9951         CK_BYTE_PTR        pPublicData2;

```

```

9952         CK_OBJECT_HANDLE      publicKey;
9953     } CK_X9_42_MQV_DERIVE_PARAMS;

```

9954

9955 The fields of the structure have the following meanings:

9956	kdf	key derivation function used on the shared secret value
9957	ulOtherInfoLen	the length in bytes of the other info
9958	pOtherInfo	some data shared between the two parties
9959	ulPublicDataLen	the length in bytes of the other party's first X9.42 Diffie-Hellman
9960		public key
9961	pPublicData	pointer to other party's first X9.42 Diffie-Hellman public key value
9962	ulPrivateDataLen	the length in bytes of the second X9.42 Diffie-Hellman private key
9963	hPrivateData	key handle for second X9.42 Diffie-Hellman private key value
9964	ulPublicDataLen2	the length in bytes of the other party's second X9.42 Diffie-Hellman
9965		public key
9966	pPublicData2	pointer to other party's second X9.42 Diffie-Hellman public key
9967		value
9968	publicKey	Handle to the first party's ephemeral public key

9969 With the key derivation function **CKD_NULL**, *pOtherInfo* must be NULL and *ulOtherInfoLen* must be zero.
9970 With the key derivation function **CKD_SHA1_KDF_ASN1**, *pOtherInfo* must be supplied, which contains
9971 an octet string, specified in ASN.1 DER encoding, consisting of mandatory and optional data shared by
9972 the two parties intending to share the shared secret. With the key derivation function
9973 **CKD_SHA1_KDF_CONCATENATE**, an optional *pOtherInfo* may be supplied, which consists of some
9974 data shared by the two parties intending to share the shared secret. Otherwise, *pOtherInfo* must be NULL
9975 and *ulOtherInfoLen* must be zero.

9976 **CK_X9_42_MQV_DERIVE_PARAMS_PTR** is a pointer to a **CK_X9_42_MQV_DERIVE_PARAMS**.

9977 6.4.12 X9.42 Diffie-Hellman key pair generation

9978 The X9.42 Diffie-Hellman key pair generation mechanism, denoted **CKM_X9_42_DH_KEY_PAIR_GEN**,
9979 is a key pair generation mechanism based on Diffie-Hellman key agreement, as defined in the [ANSI
9980 X9.42] standard.

9981 It does not have a parameter.

9982 The mechanism generates X9.42 Diffie-Hellman public/private key pairs with a particular prime, base and
9983 subprime, as specified in the **CKA_PRIME**, **CKA_BASE** and **CKA_SUBPRIME** attributes of the template
9984 for the public key.

9985 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
9986 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**, **CKA_SUBPRIME**, and
9987 **CKA_VALUE** attributes to the new private key; other attributes required by the X9.42 Diffie-Hellman
9988 public and private key types must be specified in the templates.

9989 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
9990 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

9991 6.4.13 X9.42 Diffie-Hellman domain parameter generation

9992 The X9.42 Diffie-Hellman domain parameter generation mechanism, denoted
9993 **CKM_X9_42_DH_PARAMETER_GEN**, is a domain parameters generation mechanism based on X9.42
9994 Diffie-Hellman key agreement, as defined in the [ANSI X9.42] standard.

9995 It does not have a parameter.

9996 The mechanism generates X9.42 Diffie-Hellman domain parameters with particular prime and subprime
 9997 length in bits, as specified in the **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes of the
 9998 template for the domain parameters.

9999 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PRIME**, **CKA_BASE**,
 10000 **CKA_SUBPRIME**, **CKA_PRIME_BITS** and **CKA_SUBPRIME_BITS** attributes to the new object. Other
 10001 attributes supported by the X9.42 Diffie-Hellman domain parameter types may also be specified in the
 10002 template for the domain parameters, or else are assigned default initial values.

10003 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10004 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits.

10005 6.4.14 X9.42 Diffie-Hellman key derivation

10006 The X9.42 Diffie-Hellman key derivation mechanism, denoted **CKM_X9_42_DH_DERIVE**, is a
 10007 mechanism for key derivation based on the Diffie-Hellman key agreement scheme, as defined in the
 10008 [ANSI X9.42] standard, where each party contributes one key pair, all using the same X9.42 Diffie-
 10009 Hellman domain parameters.

10010 It has a parameter, a **CK_X9_42_DH1_DERIVE_PARAMS** structure.

10011 This mechanism derives a secret value, and truncates the result according to the **CKA_KEY_TYPE**
 10012 attribute of the template and, if it has one and the key type supports it, the **CKA_VALUE_LEN** attribute of
 10013 the template. (The truncation removes bytes from the leading end of the secret value.) The mechanism
 10014 contributes the result as the **CKA_VALUE** attribute of the new key; other attributes required by the key
 10015 type must be specified in the template. Note that in order to validate this mechanism it may be required to
 10016 use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g.
 10017 **CKM_SHA_1_HMAC_GENERAL**) over some test data.

10018 This mechanism has the following rules about key sensitivity and extractability:

- 10019 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the new key can both
 10020 be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on some
 10021 default value.
- 10022 • If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE, then the derived key
 10023 will as well. If the base key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE, then the
 10024 derived key has its **CKA_ALWAYS_SENSITIVE** attribute set to the same value as its
 10025 **CKA_SENSITIVE** attribute.
- 10026 • Similarly, if the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE, then the
 10027 derived key will, too. If the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to
 10028 CK_TRUE, then the derived key has its **CKA_NEVER_EXTRACTABLE** attribute set to the *opposite*
 10029 value from its **CKA_EXTRACTABLE** attribute.

10030 When this mechanism is used in **C_EncapsulateKey** and **C_DecapsulateKey**, the mechanism
 10031 parameters *pPublicData* and *ulPublicDataLen* must be set to NULL and 0 respectively. For
 10032 **C_EncapsulateKey**, an ephemeral key pair is generated. The value of the generated public key is
 10033 returned as the ciphertext. The generated private key is used with public key provided in the API to
 10034 generate a symmetric key using Diffie Helman X9.42 Derive. For **C_DecapsulateKey**, the ciphertext is
 10035 used with the private key provided in the API to generate a symmetric key using Diffie Helman X9.42
 10036 Derive.

10037 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10038 specify the supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

10039 6.4.15 X9.42 Diffie-Hellman hybrid key derivation

10040 The X9.42 Diffie-Hellman hybrid key derivation mechanism, denoted
 10041 **CKM_X9_42_DH_HYBRID_DERIVE**, is a mechanism for key derivation based on the Diffie-Hellman
 10042 hybrid key agreement scheme, as defined in the [ANSI X9.42] standard, where each party contributes two
 10043 key pair, all using the same X9.42 Diffie-Hellman domain parameters.

10044 It has a parameter, a **CK_X9_42_DH2_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. Note that in order to validate this mechanism it may be required to use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CKM_SHA_1_HMAC_GENERAL**) over some test data.

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 supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

6.4.16 X9.42 Diffie-Hellman Menezes-Qu-Vanstone key derivation

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

It has a parameter, a **CK_X9_42_MQV_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. Note that in order to validate this mechanism it may be required to use the **CKA_VALUE** attribute as the key of a general-length MAC mechanism (e.g. **CKM_SHA_1_HMAC_GENERAL**) over some test data.

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 supported range of X9.42 Diffie-Hellman prime sizes, in bits, for the **CKA_PRIME** attribute.

6.5 Extended Triple Diffie-Hellman (x3dh)

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

Table 95, Extended Triple Diffie-Hellman Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_X3DH_INITIALIZE							✓	
CKM_X3DH_RESPOND							✓	

6.5.1 Definitions

Mechanisms:

CKM_X3DH_INITIALIZE

CKM_X3DH_RESPOND

6.5.2 Extended Triple Diffie-Hellman key objects

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

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

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

- public Identity key,
- current prekey, signed using XEDDSA with their identity key
- optionally a batch of One-time public keys.

6.5.3 Initiating an Extended Triple Diffie-Hellman key exchange

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

When the necessary key material is available, the initiating party calls **CKM_X3DH_INITIALIZE**, also providing the following additional parameters:

- the initiator's identity key
- the initiator's 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;
```

```
10127         CK_BYTE_PTR          pPrekey_signature;
10128         CK_BYTE_PTR          pOnetime_key;
10129         CK_OBJECT_HANDLE      pOwn_identity;
10130         CK_OBJECT_HANDLE      pOwn_ephemeral;
10131     } CK_X3DH_INITIATE_PARAMS;
```

10132 Table 96, Extended Triple Diffie-Hellman Initiate Message parameters:

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	Key derivation function
pPeer_identity	Key handle	Peer's public Identity key (from the prekey-bundle)
pPeer_prekey	Key Handle	Peer's 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

10133

10134 **6.5.4 Responding to an Extended Triple Diffie-Hellman key exchange**

10135 Responding an Extended Triple Diffie-Hellman key exchange is done by executing a
10136 **CKM_X3DH_RESPOND** mechanism. **CK_X3DH_RESPOND_PARAMS** is a structure that provides the
10137 parameters to the **CKM_X3DH_RESPOND** key exchange mechanism. All these parameter should be
10138 supplied by the Initiator in a message to the responder. The structure is defined as follows:

```
10139     typedef struct CK_X3DH_RESPOND_PARAMS {
10140         CK_X3DH_KDF_TYPE      kdf;
10141         CK_BYTE_PTR           pIdentity_id;
10142         CK_BYTE_PTR           pPrekey_id;
10143         CK_BYTE_PTR           pOnetime_id;
10144         CK_OBJECT_HANDLE      pInitiator_identity;
10145         CK_BYTE_PTR           pInitiator_ephemeral;
10146     } CK_X3DH_RESPOND_PARAMS;
```

10147

10148 Table 97, Extended Triple Diffie-Hellman 1st Message parameters:

Parameter	Data type	Meaning
kdf	CK_X3DH_KDF_TYPE	Key derivation function
pIdentity_id	Byte array	Peer's public Identity key identifier (from the prekey-bundle)
pPrekey_id	Byte array	Peer's 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

10149

10150 Where the *_id fields are identifiers marking which key has been used from the prekey-bundle, these
10151 identifiers could be the keys themselves.

10152
10153
10154
10155
10156
10157
10158
10159
10160
10161
10162
10163
10164

- 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.

10165 **6.5.5 Extended Triple Diffie-Hellman parameters**

- 10166 • **CK_X3DH_KDF_TYPE, CK_X3DH_KDF_TYPE_PTR**

10167 **CK_X3DH_KDF_TYPE** is used to indicate the Key Derivation Function (KDF) applied to derive keying
10168 data from a shared secret. The key derivation function will be used by the X3DH key agreement
10169 schemes. It is defined as follows:

10170 typedef CK_ULONG CK_X3DH_KDF_TYPE;

10171

10172 The following table lists the defined functions.

10173 *Table 98, X3DH: Key Derivation Functions*

Source Identifier
CKD_NULL
CKD_BLAKE2B_256_KDF
CKD_BLAKE2B_512_KDF
CKD_SHA3_256_KDF
CKD_SHA256_KDF
CKD_SHA3_512_KDF
CKD_SHA512_KDF

10174 **6.6 Double Ratchet**

10175 The Double Ratchet is a key management algorithm managing the ongoing renewal and maintenance of
10176 short-lived session keys providing forward secrecy and break-in recovery for encrypt/decrypt operations.
10177 The algorithm is described in **[DoubleRatchet]**. The Signal protocol uses X3DH to exchange a shared
10178 secret in the first step, which is then used to derive a Double Ratchet secret key.

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.

10179 Table 99, Double Ratchet Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_X2RATCHET_INITIALIZE							✓	
CKM_X2RATCHET_RESPOND							✓	
CKM_X2RATCHET_ENCRYPT	✓					✓		
CKM_X2RATCHET_DECRYPT	✓					✓		

10180

10181 6.6.1 Definitions

10182 This section defines the key type “**CKK_X2RATCHET**” for type CK_KEY_TYPE as used in the
10183 **CKA_KEY_TYPE** attribute of key objects.

10184 Mechanisms:

10185 CKM_X2RATCHET_INITIALIZE

10186 CKM_X2RATCHET_RESPOND

10187 CKM_X2RATCHET_ENCRYPT

10188 CKM_X2RATCHET_DECRYPT

10189 6.6.2 Double Ratchet secret key objects

10190 Double Ratchet secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_X2RATCHET**) hold
10191 Double Ratchet keys. Double Ratchet secret keys can only be derived from shared secret keys using the
10192 mechanism **CKM_X2RATCHET_INITIALIZE** or **CKM_X2RATCHET_RESPOND**. In the Signal protocol
10193 these are seeded with the shared secret derived from an Extended Triple Diffie-Hellman [X3DH] key-
10194 exchange. The following table defines the Double Ratchet secret key object attributes, in addition to the
10195 common attributes defined for this object class:

10196 Table 100, 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
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?

Attribute	Data type	Meaning
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:

sk	the shared secret with peer (derived using X3DH)
peers_public_prekey	Peer's public prekey which the Initiator used in the X3DH
peers_public_identity	Peer's public identity which the Initiator used in the X3DH
own_public_identity	Initiators public identity as used in the X3DH
bEncryptedHeader	whether the headers are encrypted
eCurve	255 for curve 25519 or 448 for curve 448
aeadMechanism	a mechanism supporting AEAD encryption
kdfMechanism	a Key Derivation Mechanism, such as CKD_BLAKE2B_512_KDF

- 10232
- 10233
- **CK_X2RATCHET_RESPOND_PARAMS;**
CK_X2RATCHET_RESPOND_PARAMS_PTR

10234

10235

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

10236

10237

10238

10239

10240

10241

10242

10243

10244

10245

10246

```
typedef struct CK_X2RATCHET_RESPOND_PARAMS {
    CK_BYTE_PTR          sk;
    CK_OBJECT_HANDLE     own_prekey;
    CK_OBJECT_HANDLE     initiator_identity;
    CK_OBJECT_HANDLE     own_public_identity;
    CK_BBOOL             bEncryptedHeader;
    CK_ULONG             eCurve;
    CK_MECHANISM_TYPE    aeadMechanism;
    CK_X2RATCHET_KDF_TYPE kdfMechanism;
} CK_X2RATCHET_RESPOND_PARAMS;
```

10247

The fields of the structure have the following meanings:

10248

10249

10250

10251

10252

10253

10254

10255

sk	shared secret with the Initiator
own_prekey	Own Prekey pair that the Initiator used
initiator_identity	Initiator's public identity key used
own_public_identity	as used in the prekey bundle by the initiator in the X3DH
bEncryptedHeader	whether the headers are encrypted
eCurve	255 for curve 25519 or 448 for curve 448
aeadMechanism	a mechanism supporting AEAD encryption
kdfMechanism	a Key Derivation Mechanism, such as CKD_BLAKE2B_512_KDF

10256

6.6.4 Double Ratchet Encryption mechanism

10257

10258

10259

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

10260

6.6.5 Double Ratchet parameters

- 10261
- **CK_X2RATCHET_KDF_TYPE, CK_X2RATCHET_KDF_TYPE_PTR**

10262

10263

10264

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:

10265

10266

```
typedef CK_ULONG CK_X2RATCHET_KDF_TYPE;
```

10267

The following table lists the defined functions.

10268

Table 101, 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

10269

10270 6.7 Wrapping/unwrapping private keys

10271 Cryptoki Versions 2.01 and up allow the use of secret keys for wrapping and unwrapping RSA private
 10272 keys, Diffie-Hellman private keys, X9.42 Diffie-Hellman private keys, short Weierstrass EC private keys,
 10273 DSA private keys, HSS private keys, XMSS/XMSSMT private keys, ML-DSA private keys, ML-KEM
 10274 private keys and SLH-DSA private keys. For wrapping, a private key is BER-encoded according to [PKCS
 10275 #8] PrivateKeyInfo ASN.1 type. [PKCS #8] requires an algorithm identifier for the type of the private key.
 10276 The object identifiers for the required algorithm identifiers are as follows:

```

10277     rsaEncryption OBJECT IDENTIFIER ::= { pkcs-1 1 }
10278
10279     dhKeyAgreement OBJECT IDENTIFIER ::= { pkcs-3 1 }
10280
10281     dhpublicnumber OBJECT IDENTIFIER ::= { iso(1) member-body(2)
10282         us(840) ansi-x942(10046) number-type(2) 1 }
10283
10284     id-ecPublicKey OBJECT IDENTIFIER ::= { iso(1) member-body(2)
10285         us(840) ansi-x9-62(10045) publicKeyType(2) 1 }
10286
10287     id-dsa OBJECT IDENTIFIER ::= { iso(1) member-body(2) us(840)
10288         x9-57(10040) x9cm(4) 1 }
10289
10290     id-ml-dsa-44 OBJECT IDENTIFIER ::= { nist-sigAlgs 17 }
10291     id-ml-dsa-65 OBJECT IDENTIFIER ::= { nist-sigAlgs 18 }
10292     id-ml-dsa-87 OBJECT IDENTIFIER ::= { nist-sigAlgs 19 }
10293
10294     id-slh-dsa-sha2-128s OBJECT IDENTIFIER ::= { nist-sigAlgs 20
10295         }
10296     id-slh-dsa-sha2-128f OBJECT IDENTIFIER ::= { nist-sigAlgs 21
10297         }
10298     id-slh-dsa-sha2-192s OBJECT IDENTIFIER ::= { nist-sigAlgs 22
10299         }
10300     id-slh-dsa-sha2-192f OBJECT IDENTIFIER ::= { nist-sigAlgs 23
10301         }
10302     id-slh-dsa-sha2-256s OBJECT IDENTIFIER ::= { nist-sigAlgs 24
10303         }
10304     id-slh-dsa-sha2-256f OBJECT IDENTIFIER ::= { nist-sigAlgs 25
10305         }
10306     id-slh-dsa-shake-128s OBJECT IDENTIFIER ::= { nist-sigAlgs 26
10307         }
10308     id-slh-dsa-shake-128f OBJECT IDENTIFIER ::= { nist-sigAlgs 27
10309         }
10310     id-slh-dsa-shake-192s OBJECT IDENTIFIER ::= { nist-sigAlgs 28

```

```

10311     }
10312     id-slh-dsa-shake-192f OBJECT IDENTIFIER ::= { nist-sigAlgs 29
10313     }
10314     id-slh-dsa-shake-256s OBJECT IDENTIFIER ::= { nist-sigAlgs 30
10315     }
10316     id-slh-dsa-shake-256f OBJECT IDENTIFIER ::= { nist-sigAlgs 31
10317     }
10318
10319     id-hash-ml-dsa-44-with-sha512 OBJECT IDENTIFIER ::= {
10320         nist-sigAlgs 32 }
10321     id-hash-ml-dsa-65-with-sha512 OBJECT IDENTIFIER ::= {
10322         nist-sigAlgs 33 }
10323     id-hash-ml-dsa-87-with-sha512 OBJECT IDENTIFIER ::= {
10324         nist-sigAlgs 34 }
10325
10326     id-hash-slh-dsa-sha2-128s-with-sha256 OBJECT IDENTIFIER ::= {
10327         nist-sigAlgs 35 }
10328     id-hash-slh-dsa-sha2-128f-with-sha256 OBJECT IDENTIFIER ::= {
10329         nist-sigAlgs 36 }
10330     id-hash-slh-dsa-sha2-192s-with-sha512 OBJECT IDENTIFIER ::= {
10331         nist-sigAlgs 37 }
10332     id-hash-slh-dsa-sha2-192f-with-sha512 OBJECT IDENTIFIER ::= {
10333         nist-sigAlgs 38 }
10334     id-hash-slh-dsa-sha2-256s-with-sha512 OBJECT IDENTIFIER ::= {
10335         nist-sigAlgs 39 }
10336     id-hash-slh-dsa-sha2-256f-with-sha512 OBJECT IDENTIFIER ::= {
10337         nist-sigAlgs 40 }
10338     id-hash-slh-dsa-shake-128s-with-shake128 OBJECT IDENTIFIER
10339         ::= { nist-sigAlgs 41 }
10340     id-hash-slh-dsa-shake-128f-with-shake128 OBJECT IDENTIFIER
10341         ::= { nist-sigAlgs 42 }
10342     id-hash-slh-dsa-shake-192s-with-shake256 OBJECT IDENTIFIER
10343         ::= { nist-sigAlgs 43 }
10344     id-hash-slh-dsa-shake-192f-with-shake256 OBJECT IDENTIFIER
10345         ::= { nist-sigAlgs 44 }
10346     id-hash-slh-dsa-shake-256s-with-shake256 OBJECT IDENTIFIER
10347         ::= { nist-sigAlgs 45 }
10348     id-hash-slh-dsa-shake-256f-with-shake256 OBJECT IDENTIFIER
10349         ::= { nist-sigAlgs 46 }
10350
10351     id-alg-ml-kem-512 OBJECT IDENTIFIER ::= { nist-kems 1 }
10352     id-alg-ml-kem-768 OBJECT IDENTIFIER ::= { nist-kems 2 }
10353     id-alg-ml-kem-1024 OBJECT IDENTIFIER ::= { nist-kems 3 }
10354
10355     where
10356     pkcs-1 OBJECT IDENTIFIER ::= {
10357         iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 1 }
10358

```

```

10359 pkcs-3 OBJECT IDENTIFIER ::= {
10360     iso(1) member-body(2) US(840) rsadsi(113549) pkcs(1) 3 }
10361
10362 nistAlgorithms OBJECT IDENTIFIER ::= { joint-iso-ccitt(2)
10363     country(16) us(840) organization(1) gov(101) csor(3)
10364     nistAlgorithm(4) }
10365 nist-sigAlgs OBJECT IDENTIFIER ::= { nistAlgorithms 3 }
10366 nist-kems OBJECT IDENTIFIER ::= { nistAlgorithms 4 }
10367
10368 These parameters for the algorithm identifiers have the
10369     following types, respectively:
10370 NULL
10371
10372 DHParameter ::= SEQUENCE {
10373     prime          INTEGER, -- p
10374     base           INTEGER, -- g
10375     privateValueLength  INTEGER OPTIONAL
10376 }
10377
10378 DomainParameters ::= SEQUENCE {
10379     prime          INTEGER, -- p
10380     base           INTEGER, -- g
10381     subprime       INTEGER, -- q
10382     cofactor       INTEGER OPTIONAL, -- j
10383     validationParms ValidationParms OPTIONAL
10384 }
10385
10386 ValidationParms ::= SEQUENCE {
10387     Seed           BIT STRING, -- seed
10388     PGenCounter    INTEGER      -- parameter verification
10389 }
10390
10391 Parameters ::= CHOICE {
10392     ecParameters    ECParameters,
10393     namedCurve       CURVES.&id({CurveNames}),
10394     implicitlyCA     NULL
10395 }
10396
10397 Dss-Parms ::= SEQUENCE {
10398     p INTEGER,
10399     q INTEGER,
10400     g INTEGER
10401 }
10402

```

10403 For the X9.42 Diffie-Hellman domain parameters, the **cofactor** and the **validationParms** optional fields
10404 should not be used when wrapping or unwrapping X9.42 Diffie-Hellman private keys since their values
10405 are not stored within the token.

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

Within the PrivateKeyInfo type:

- RSA private keys are BER-encoded according to PKCS #1's RSAPrivateKey ASN.1 type. This type requires values to be present for *all* the attributes specific to Cryptoki's RSA private key objects. In other words, if a Cryptoki library does not have values for an RSA private key's **CKA_MODULUS**, **CKA_PUBLIC_EXPONENT**, **CKA_PRIVATE_EXPONENT**, **CKA_PRIME_1**, **CKA_PRIME_2**, **CKA_EXPONENT_1**, **CKA_EXPONENT_2**, and **CKA_COEFFICIENT** values, it must not create an RSAPrivateKey BER-encoding of the key, and so it must not prepare it for wrapping.
- Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
- X9.42 Diffie-Hellman private keys are represented as BER-encoded ASN.1 type INTEGER.
- Short Weierstrass EC private keys are BER-encoded according to SECG [SEC 1] ECPrivateKey ASN.1 type:

```
ECPrivateKey ::= SEQUENCE {  
    Version          INTEGER { ecPrivkeyVer1(1) }  
        (ecPrivkeyVer1),  
    privateKey       OCTET STRING,  
    parameters       [0] Parameters OPTIONAL,  
    publicKey        [1] BIT STRING OPTIONAL  
}
```

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

- DSA private keys are represented as BER-encoded ASN.1 type INTEGER.
- For ML-KEM and ML-DSA keys, the private key can be encoded with a seed, a value, or both. If only one of the two are supplied, unwrap may fail in some tokens. Which encodings on wrap are specified is token specific.

Once a private key has been BER-encoded as a PrivateKeyInfo type, the resulting string of bytes is encrypted with the secret key. This encryption is defined in the section for the respective key wrapping mechanism.

Unwrapping a wrapped private key undoes the above procedure. The ciphertext is decrypted as defined for the respective key unwrapping mechanism. The data thereby obtained are parsed as a PrivateKeyInfo type. An error will result if the original wrapped key does not decrypt properly, or if the decrypted data does not parse properly, or its type does not match the key type specified in the template for the new key. The unwrapping mechanism contributes only those attributes specified in the PrivateKeyInfo type to the newly-unwrapped key; other attributes must be specified in the template, or will take their default values.

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

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

with associated parameters

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

```
10454         base INTEGER -- base g
10455     }
10456
```

10457 for wrapping DSA private keys. Note that although the two structures for holding DSA domain parameters
10458 appear identical when instances of them are encoded, the two corresponding object identifiers are
10459 different.

10460 **6.8 Generic secret key**

10461 *Table 102, Generic Secret Key Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_GENERIC_SECRET_KEY_GEN					✓			

10462 **6.8.1 Definitions**

10463 This section defines the key type “**CKK_GENERIC_SECRET**” for type CK_KEY_TYPE as used in the
10464 **CKA_KEY_TYPE** attribute of key objects.

10465 Mechanisms:

```
10466         CKM_GENERIC_SECRET_KEY_GEN
```

10467 **6.8.2 Generic secret key objects**

10468 Generic secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_GENERIC_SECRET**) hold
10469 generic secret keys. These keys do not support encryption or decryption; however, other keys can be
10470 derived from them and they can be used in HMAC operations. The following table defines the generic
10471 secret key object attributes, in addition to the common attributes defined for this object class:

10472 These key types are used in several of the mechanisms described in this section.

10473 *Table 103, 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

10474 Refer to Table 13 for footnotes

10475 The following is a sample template for creating a generic secret key object:

```
10476     CK_OBJECT_CLASS class = CKO_SECRET_KEY;
10477     CK_KEY_TYPE keyType = CKK_GENERIC_SECRET;
10478     CK_UTF8CHAR label[] = "A generic secret key object";
10479     CK_BYTE value[] = {...};
10480     CK_BBOOL true = CK_TRUE;
10481     CK_ATTRIBUTE template[] = {
10482         {CKA_CLASS, &class, sizeof(class)},
10483         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
10484         {CKA_TOKEN, &true, sizeof(true)},
```

```

10485     {CKA_LABEL, label, sizeof(label)-1},
10486     {CKA_DERIVE, &true, sizeof(true)},
10487     {CKA_VALUE, value, sizeof(value)}
10488 };

```

10489

10490 **CKA_CHECK_VALUE**: The value of this attribute is derived from the key object by taking the first three
 10491 bytes of the SHA-1 hash of the generic secret key object's **CKA_VALUE** attribute.

10492 6.8.3 Generic secret key generation

10493 The generic secret key generation mechanism, denoted **CKM_GENERIC_SECRET_KEY_GEN**, is used
 10494 to generate generic secret keys. The generated keys take on any attributes provided in the template
 10495 passed to the **C_GenerateKey** call, and the **CKA_VALUE_LEN** attribute specifies the length of the key
 10496 to be generated.

10497 It does not have a parameter.

10498 The template supplied must specify a value for the **CKA_VALUE_LEN** attribute. If the template specifies
 10499 an object type and a class, they must have the following values:

```

10500     CK_OBJECT_CLASS = CKO_SECRET_KEY;
10501     CK_KEY_TYPE = CKK_GENERIC_SECRET;

```

10502 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
 10503 specify the supported range of key sizes, in bits.

10504 6.9 HMAC mechanisms

10505 Refer to [RFC 2104] and [FIPS 198] for HMAC algorithm description. The HMAC secret key shall
 10506 correspond to the PKCS11 generic secret key type or the mechanism specific key types (see mechanism
 10507 definition). Such keys, for use with HMAC operations can be created using **C_CreateObject**,
 10508 **C_GenerateKey**, or **C_UnwrapKey**.

10509 The RFC also specifies test vectors for the various hash function based HMAC mechanisms described in
 10510 the respective hash mechanism descriptions. The RFC should be consulted to obtain these test vectors.

10511 6.9.1 General block cipher mechanism parameters

10512 • **CK_MAC_GENERAL_PARAMS; CK_MAC_GENERAL_PARAMS_PTR**

10513 **CK_MAC_GENERAL_PARAMS** provides the parameters to the general-length MACing mechanisms of
 10514 the DES, DES3 (triple-DES), AES, Camellia, SEED, and ARIA ciphers. It also provides the parameters to
 10515 the general-length HMACing mechanisms (i.e., SHA-1, SHA-256, SHA-384, SHA-512, and SHA-512/T
 10516 family) and the two SSL 3.0 MACing mechanisms, (i.e., MD5 and SHA-1). It holds the length of the MAC
 10517 that these mechanisms produce. It is defined as follows:

```

10518     typedef CK_ULONG CK_MAC_GENERAL_PARAMS;

```

10519

10520 **CK_MAC_GENERAL_PARAMS_PTR** is a pointer to a **CK_MAC_GENERAL_PARAMS**.

10521 6.10 AES

10522 For the Advanced Encryption Standard (AES) see [FIPS PUB 197].

10523 *Table 104, AES Mechanisms vs. Functions*

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

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:

10545 Table 105, 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}	CK_ULONG	Length in bytes of key value

10546 Refer to Table 13 for footnotes

10547 The following is a sample template for creating an AES secret key object:

```
10548 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
10549 CK_KEY_TYPE keyType = CKK_AES;
10550 CK_UTF8CHAR label[] = "An AES secret key object";
10551 CK_BYTE value[] = {...};
10552 CK_BBOOL true = CK_TRUE;
10553 CK_ATTRIBUTE template[] = {
10554     {CKA_CLASS, &class, sizeof(class)},
10555     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
10556     {CKA_TOKEN, &true, sizeof(true)},
10557     {CKA_LABEL, label, sizeof(label)-1},
10558     {CKA_ENCRYPT, &true, sizeof(true)},
10559     {CKA_VALUE, value, sizeof(value)}
10560 };
```

10561

10562
10563
10564

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

10565 **6.10.3 AES key generation**

10566 The AES key generation mechanism, denoted **CKM_AES_KEY_GEN**, is a key generation mechanism for
10567 NIST's Advanced Encryption Standard.

10568 It does not have a parameter.

10569 The mechanism generates AES keys with a particular length in bytes, as specified in the
10570 **CKA_VALUE_LEN** attribute of the template for the key.

10571 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
10572 key. Other attributes supported by the AES key type (specifically, the flags indicating which functions the
10573 key supports) may be specified in the template for the key, or else are assigned default initial values.

10574 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10575 specify the supported range of AES key sizes, in bytes.

10576 **6.10.4 AES-ECB**

10577 AES-ECB, denoted **CKM_AES_ECB**, is a mechanism for single- and multiple-part encryption and
10578 decryption; key wrapping; and key unwrapping, based on NIST Advanced Encryption Standard and
10579 electronic codebook mode.

10580 It does not have a parameter.

10581 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
10582 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
10583 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
10584 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 106, 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	

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

6.10.5 AES-CBC

AES-CBC, denoted **CKM_AES_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced Encryption Standard 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:

10611 Table 107, 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	

10612 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10613 specify the supported range of AES key sizes, in bytes.

10614 **6.10.6 AES-CBC with PKCS padding**

10615 AES-CBC with PKCS padding, denoted **CKM_AES_CBC_PAD**, is a mechanism for single- and multiple-
10616 part encryption and decryption; key wrapping; and key unwrapping, based on NIST's Advanced
10617 Encryption Standard; cipher-block chaining mode; and the block cipher padding method detailed in
10618 [PKCS #7].

10619 It has a parameter, a 16-byte initialization vector.

10620 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
10621 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
10622 the **CKA_VALUE_LEN** attribute.

10623 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
10624 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see section 6.7 for
10625 details). The entries in the table below for data length constraints when wrapping and unwrapping keys do
10626 not apply to wrapping and unwrapping private keys.

10627 Constraints on key types and the length of data are summarized in the following table:

10628 Table 108, 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

10629 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
10630 specify the supported range of AES key sizes, in bytes.

10631 **6.10.7 AES-OFB**

10632 AES-OFB, denoted **CKM_AES_OFB**. It is a mechanism for single and multiple-part encryption and
10633 decryption with AES. AES-OFB mode is described in [NIST sp800-38a].

10634 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
10635 the block size.

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

Table 109, 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

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

6.10.8 AES-CFB

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

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 110, 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 111, 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 112, 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 113, 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

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

6.10.12 AES-XCBC-MAC-96

AES-XCBC-MAC-96, denoted **CKM_AES_XCBC_MAC_96**, 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 114, 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

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

6.11 AES with Counter

Table 115, AES with Counter Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_AES_CTR	✓					✓		

6.11.1 Definitions

Mechanisms:

CKM_AES_CTR

10693

10694

10695

10696

10697

10698

10699

10700

10702

10704

10705

10707

10708

10709

10710

10711
1071210712
10712

10714

10715

10716

10717

10718

10720

10721

10722

10723

10724

10726

10728

10730

10734

10736

10737

10738 This mode allows unpadded data that has length that is not a multiple of the block size to be encrypted to
10739 the same length of ciphertext.

10740 Table 116, AES CBC with CipherText Stealing CTS Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_AES_CTS	✓					✓		

10741 6.12.1 Definitions

10742 Mechanisms:

10743 CKM_AES_CTS

10744 6.12.2 AES CTS mechanism parameters

10745 It has a parameter, a 16-byte initialization vector.

10746 Table 117, AES-CTS: Key And Data Length

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

10747

10748 6.13 Additional AES Mechanisms

10749 Table 118, Additional AES Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_AES_GCM	✓					✓		
CKM_AES_CCM	✓					✓		
CKM_AES_GMAC		✓						

10750

10751 6.13.1 Definitions

10752 Mechanisms:

10753 CKM_AES_GCM

10754 CKM_AES_CCM

10755 CKM_AES_GMAC

10756 Generator Functions:

10757 CKG_NO_GENERATE
10758 CKG_GENERATE
10759 CKG_GENERATE_COUNTER
10760 CKG_GENERATE_RANDOM
10761 CKG_GENERATE_COUNTER_XOR

10762 6.13.2 AES-GCM Authenticated Encryption / Decryption

10763 Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where *K*
10764 (key) and AAD (additional authenticated data) are as described in [GCM]. AES-GCM uses
10765 CK_GCM_PARAMS for Encrypt, Decrypt and CK_GCM_MESSAGE_PARAMS for MessageEncrypt and
10766 MessageDecrypt.

10767 Encrypt:

- 10768 • Set the IV length *ullvLen* in the parameter block.
- 10769 • Set the IV data *pIv* in the parameter block.
- 10770 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
10771 *ulAADLen* is 0.
- 10772 • Set the tag length *ulTagBits* in the parameter block.
- 10773 • Call **C_EncryptInit** for **CKM_AES_GCM** mechanism with parameters and key *K*.
- 10774 • Call **C_Encrypt**, or **C_EncryptUpdate**^{*4} **C_EncryptFinal**, for the plaintext obtaining ciphertext and
10775 authentication tag output.

10776 Decrypt:

- 10777 • Set the IV length *ullvLen* in the parameter block.
- 10778 • Set the IV data *pIv* in the parameter block.
- 10779 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
10780 *ulAADLen* is 0.
- 10781 • Set the tag length *ulTagBits* in the parameter block.
- 10782 • Call **C_DecryptInit** for **CKM_AES_GCM** mechanism with parameters and key *K*.
- 10783 • Call **C_Decrypt**, or **C_DecryptUpdate**^{*1} **C_DecryptFinal**, for the ciphertext, including the
10784 appended tag, obtaining plaintext output. Note: since **CKM_AES_GCM** is an AEAD cipher, no data
10785 should be returned until **C_Decrypt** or **C_DecryptFinal**.

10786 MessageEncrypt:

- 10787 • Set the IV length *ullvLen* in the parameter block.
- 10788 • Set *pIv* to hold the IV data returned from **C_EncryptMessage** and **C_EncryptMessageBegin**. If
10789 *ullvFixedBits* is not zero, then the *ullvFixedBits* most significant bits of *pIv* contain the fixed IV. If
10790 *ivGenerator* is set to **CKG_NO_GENERATE**, *pIv* is an input parameter with the full IV.
- 10791 • Set the *ullvFixedBits* and *ivGenerator* fields in the parameter block.
- 10792 • Set the tag length *ulTagBits* in the parameter block.
- 10793 • Set *pTag* to hold the tag data returned from **C_EncryptMessage** or the final
10794 **C_EncryptMessageNext**.

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

10795 • Call **C_MessageEncryptInit** for **CKM_AES_GCM** mechanism key *K*.

10796 • Call **C_EncryptMessage**, or **C_EncryptMessageBegin** followed by **C_EncryptMessageNext**^{*5}.

10797 The mechanism parameter is passed to all three of these functions.

10798 • Call **C_MessageEncryptFinal** to close the message encryption.

10799 MessageDecrypt:

10800 • Set the IV length *ullvLen* in the parameter block.

10801 • Set the IV data *pIv* in the parameter block.

10802 • The *ullvFixedBits* and *ivGenerator* fields are ignored.

10803 • Set the tag length *ulTagBits* in the parameter block.

10804 • Set the tag data *pTag* in the parameter block before **C_DecryptMessage** or the final

10805 **C_DecryptMessageNext**.

10806 • Call **C_MessageDecryptInit** for **CKM_AES_GCM** mechanism key *K*.

10807 • Call **C_DecryptMessage**, or **C_DecryptMessageBegin** followed by **C_DecryptMessageNext**^{*6}.

10808 The mechanism parameter is passed to all three of these functions.

10809 • Call **C_MessageDecryptFinal** to close the message decryption.

10810 In *pIv* the least significant bit of the initialization vector is the rightmost bit. *ullvLen* is the length of the

10811 initialization vector in bytes.

10812 On MessageEncrypt, the meaning of *ivGenerator* is as follows:

10813 **CKG_NO_GENERATE** means the IV is passed in on MessageEncrypt and no internal IV generation is

10814 done.

10815 **CKG_GENERATE** means that the non-fixed portion of the IV is generated by the module internally. The

10816 generation method is not defined.

10817 **CKG_GENERATE_COUNTER** means that the non-fixed portion of the IV is generated by the module

10818 internally by use of an incrementing counter, the initial IV counter is zero.

10819 **CKG_GENERATE_COUNTER_XOR** means that the non-fixed portion of the IV is xored with a counter.

10820 The value of the non-fixed portion passed must not vary from call to call. Like

10821 **CKG_GENERATE_COUNTER**, the counter starts at zero.

10822 **CKG_GENERATE_RANDOM** means that the non-fixed portion of the IV is generated by the module

10823 internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in *pIV*.

10824 Modules must implement **CKG_GENERATE**. Modules may also reject *ullvFixedBits* values which are too

10825 large. Zero is always an acceptable value for *ullvFixedBits*.

10826 In Encrypt and Decrypt the tag is appended to the ciphertext and the least significant bit of the tag is the

10827 rightmost bit and the tag bits are the rightmost *ulTagBits* bits. In MessageEncrypt the tag is returned in

10828 the *pTag* field of CK_GCM_MESSAGE_PARAMS. In MessageDecrypt the tag is provided by the *pTag*

10829 field of CK_GCM_MESSAGE_PARAMS.

10830 The key type for *K* must be compatible with **CKM_AES_ECB** and the

10831 **C_EncryptInit/C_DecryptInit/C_MessageEncryptInit/C_MessageDecryptInit** calls shall behave, with

10832 respect to *K*, as if they were called directly with **CKM_AES_ECB**, *K* and NULL parameters.

⁵ "*" indicates 0 or more calls may be made as required

⁶ "*" indicates 0 or more calls may be made as required

6.13.3 AES-GCM Authenticated Wrap / Unwrap

Generic GCM mode is described in [GCM]. To set up for AES-GCM use the following process, where *wK* (wrapping key) and *AAD* (additional authenticated data) are as described in [GCM]. AES-GCM uses **CK_GCM_WRAP_PARAMS** for *WrapKey*, *UnwrapKey* and **CK_GCM_MESSAGE_PARAMS** for *WrapKeyAuthenticated* and *UnwrapKeyAuthenticated*.

WrapKey:

- Set the IV length *ullvLen* in the parameter block.
- Set *pIv* to hold the IV data returned from **C_WrapKey**. If *ullvFixedBits* is not zero, then the *ullvFixedBits* most significant bits of *pIv* contain the fixed IV. If *ivGenerator* is set to **CKG_NO_GENERATE**, *pIv* is an input parameter with the full IV.
- Set the *ullvFixedBits* and *ivGenerator* fields in the parameter block.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- Set the tag length *ulTagBits* in the parameter block.
- Call **C_WrapKey** for **CKM_AES_GCM** mechanism with parameters and wrapping key *wK* and key to be wrapped *K*, obtaining a wrapped key and authentication tag output.

UnwrapKey:

- Set the IV length *ullvLen* in the parameter block.
- Set the IV data *pIv* in the parameter block.
- The *ullvFixedBits* and *ivGenerator* fields are ignored.
- Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if *ulAADLen* is 0.
- Set the tag length *ulTagBits* in the parameter block.
- Call **C_UnwrapKey** for **CKM_AES_GCM** mechanism with parameters and wrapping key *K*, wrapped key and authenticated tag output from wrap, and template for the new key, obtaining a key handle.

WrapKeyAuthenticated:

- Set the IV length *ullvLen* in the parameter block.
- Set *pIv* to hold the IV data returned from **C_WrapKeyAuthenticated**. If *ullvFixedBits* is not zero, then the *ullvFixedBits* most significant bits of *pIv* contain the fixed IV. If *ivGenerator* is set to **CKG_NO_GENERATE**, *pIv* is an input parameter with the full IV.
- Set the *ullvFixedBits* and *ivGenerator* fields in the parameter block.
- Set the tag length *ulTagBits* in the parameter block.
- Set *pTag* to hold the tag data returned from **C_WrapKeyAuthenticated**.
- Call **C_WrapKeyAuthenticated** for **CKM_AES_GCM** mechanism wrapping key *wK*, wrapped key, parameter and obtaining a wrapped key and authentication tag output in the parameter block.

UnwrapKeyAuthenticated:

- Set the IV length *ullvLen* in the parameter block.
- Set the IV data *pIv* in the parameter block.
- The *ullvFixedBits* and *ivGenerator* fields are ignored.
- Set the tag length *ulTagBits* in the parameter block.

10874 • Set the tag data *pTag* in the parameter block.

10875 • Call **C_UnwrapKeyAuthenticated** for **CKM_AES_GCM** mechanism, wrapping key *wK*, wrapped

10876 key and authenticated tag output from wrap, and template for the new key, obtaining a key handle.

10877 In *plv* the least significant bit of the initialization vector is the rightmost bit. *ullvLen* is the length of the

10878 initialization vector in bytes.

10879 On **WrapKeyAuthenticated**, the meaning of *ivGenerator* is as follows:

10880 **CKG_NO_GENERATE** means the IV is passed in on **WrapKeyAuthenticated** and no internal IV generation

10881 is done.

10882 **CKG_GENERATE** means that the non-fixed portion of the IV is generated by the module internally. The

10883 generation method is not defined.

10884 **CKG_GENERATE_COUNTER** means that the non-fixed portion of the IV is generated by the module

10885 internally by use of an incrementing counter, the initial IV counter is zero.

10886 **CKG_GENERATE_COUNTER_XOR** means that the non-fixed portion of the IV is xored with a counter.

10887 The value of the non-fixed portion passed must not vary from call to call. Like

10888 **CKG_GENERATE_COUNTER**, the counter starts at zero.

10889 **CKG_GENERATE_RANDOM** means that the non-fixed portion of the IV is generated by the module

10890 internally using a PRNG. In any case the entire IV, including the fixed portion, is returned in *pIV*.

10891 Modules must implement **CKG_GENERATE**. Modules may also reject *ullvFixedBits* values which are too

10892 large. Zero is always an acceptable value for *ullvFixedBits*.

10893 In **WrapKey** and **UnwrapKey** the tag is appended to the wrapped key and the least significant bit of the tag

10894 is the rightmost bit and the tag bits are the rightmost *ulTagBits* bits. In **WrapKeyAuthenticated** the tag is

10895 returned in the *pTag* field of **CK_GCM_MESSAGE_PARAMS**. In **UnwrapKeyAuthenticated** the tag is

10896 provided by the *pTag* field of **CK_GCM_MESSAGE_PARAMS**.

10897 6.13.4 AES-CCM Authenticated Encryption / Decryption

10898 For IPsec ([RFC 4309]) and also for use in ZFS encryption. Generic CCM mode is described in [RFC

10899 3610].

10900 To set up for AES-CCM use the following process, where *K* (key), nonce and additional authenticated

10901 data are as described in [RFC 3610]. AES-CCM uses **CK_CCM_PARAMS** for Encrypt and Decrypt, and

10902 **CK_CCM_MESSAGE_PARAMS** for **MessageEncrypt** and **MessageDecrypt**.

10903 Encrypt:

10904 • Set the message/data length *ulDataLen* in the parameter block.

10905 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.

10906 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if

10907 *ulAADLen* is 0.

10908 • Set the MAC length *ulMACLen* in the parameter block.

10909 • Call **C_EncryptInit** for **CKM_AES_CCM** mechanism with parameters and key *K*.

10910 • Call **C_Encrypt**, **C_EncryptUpdate**, or **C_EncryptFinal**, for the plaintext obtaining the final

10911 ciphertext output and the MAC. The total length of data processed must be *ulDataLen*. The output

10912 length will be *ulDataLen* + *ulMACLen*.

10913 Decrypt:

10914 • Set the message/data length *ulDataLen* in the parameter block. This length must not include the

10915 length of the MAC that is appended to the ciphertext.

10916 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.

10917 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
10918 *ulAADLen* is 0.

10919 • Set the MAC length *ulMACLen* in the parameter block.

10920 • Call **C_DecryptInit** for **CKM_AES_CCM** mechanism with parameters and key *K*.

10921 • Call **C_Decrypt**, **C_DecryptUpdate**, or **C_DecryptFinal**, for the ciphertext, including the
10922 appended MAC, obtaining plaintext output. The total length of data processed must be *ulDataLen*
10923 + *ulMACLen*. Note: since **CKM_AES_CCM** is an AEAD cipher, no data should be returned until
10924 **C_Decrypt** or **C_DecryptFinal**.

10925 MessageEncrypt:

10926 • Set the message/data length *ulDataLen* in the parameter block.

10927 • Set the nonce length *ulNonceLen*.

10928 • Set *pNonce* to hold the nonce data returned from **C_EncryptMessage** and
10929 **C_EncryptMessageBegin**. If *ulNonceFixedBits* is not zero, then the most significant bits of
10930 *pNonce* contain the fixed nonce. If *nonceGenerator* is set to **CKG_NO_GENERATE**, *pNonce* is an
10931 input parameter with the full nonce.

10932 • Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.

10933 • Set the MAC length *ulMACLen* in the parameter block.

10934 • Set *pMAC* to hold the MAC data returned from **C_EncryptMessage** or the final
10935 **C_EncryptMessageNext**.

10936 • Call **C_MessageEncryptInit** for **CKM_AES_CCM** mechanism key *K*.

10937 • Call **C_EncryptMessage**, or **C_EncryptMessageBegin** followed by **C_EncryptMessageNext**^{*7}.
10938 The mechanism parameter is passed to all three functions.

10939 • Call **C_MessageEncryptFinal** to close the message encryption.

10940 • The MAC is returned in *pMac* of the CK_CCM_MESSAGE_PARAMS structure.

10941 MessageDecrypt:

10942 • Set the message/data length *ulDataLen* in the parameter block.

10943 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block

10944 • The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.

10945 • Set the MAC length *ulMACLen* in the parameter block.

10946 • Set the MAC data *pMAC* in the parameter block before **C_DecryptMessage** or the final
10947 **C_DecryptMessageNext**.

10948 • Call **C_MessageDecryptInit** for **CKM_AES_CCM** mechanism key *K*.

10949 • Call **C_DecryptMessage**, or **C_DecryptMessageBegin** followed by **C_DecryptMessageNext**^{*8}.
10950 The mechanism parameter is passed to all three functions.

10951 • Call **C_MessageDecryptFinal** to close the message decryption.

10952 In *pNonce* the least significant bit of the nonce is the rightmost bit. *ulNonceLen* is the length of the nonce
10953 in bytes.

⁷ "*" indicates 0 or more calls may be made as required

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

10954 On MessageEncrypt, the meaning of *nonceGenerator* is as follows:

10955 **CKG_NO_GENERATE** means the nonce is passed in on MessageEncrypt and no internal nonce

10956 generation is done.

10957 **CKG_GENERATE** means that the non-fixed portion of the nonce is generated by the module internally.

10958 The generation method is not defined.

10959 **CKG_GENERATE_COUNTER** means that the non-fixed portion of the nonce is generated by the module

10960 internally by use of an incrementing counter, the initial IV counter is zero.

10961 **CKG_GENERATE_COUNTER_XOR** means that the non-fixed portion of the IV is xored with a counter.

10962 The value of the non-fixed portion passed must not vary from call to call. Like

10963 **CKG_GENERATE_COUNTER**, the counter starts at zero.

10964 **CKG_GENERATE_RANDOM** means that the non-fixed portion of the nonce is generated by the module

10965 internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.

10966 Modules must implement **CKG_GENERATE**. Modules may also reject *ulNonceFixedBits* values which

10967 are too large. Zero is always an acceptable value for *ulNonceFixedBits*.

10968 In Encrypt and Decrypt the MAC is appended to the ciphertext and the least significant byte of the MAC is

10969 the rightmost byte and the MAC bytes are the rightmost *ulMACLen* bytes. In MessageEncrypt the MAC is

10970 returned in the *pMAC* field of CK_CCM_MESSAGE_PARAMS. In MessageDecrypt the MAC is provided

10971 by the *pMAC* field of CK_CCM_MESSAGE_PARAMS.

10972 The key type for K must be compatible with **CKM_AES_ECB** and the

10973 **C_EncryptInit/C_DecryptInit/C_MessageEncryptInit/C_MessageDecryptInit** calls shall behave, with

10974 respect to K, as if they were called directly with **CKM_AES_ECB**, K and NULL parameters.

10975 6.13.5 AES-CCM Authenticated Wrap / Unwrap

10976 To set up for AES-CCM use the following process, where *K* (key), nonce and additional authenticated

10977 data are as described in [RFC 3610]. AES-CCM uses CK_CCM_WRAP_PARAMS for WrapKey and

10978 UnwrapKey, and CK_CCM_MESSAGE_PARAMS for WrapKeyAuthenticated and

10979 UnwrapKeyAuthenticated.

10980 WrapKey:

- 10981 • Set the message/data length *ulDataLen* in the parameter block to zero. The token will determine
- 10982 the message/data length during the wrap operation based on the key being wrapped. Previous
- 10983 versions of the specification state that the application should populate this field. As such, tokens
- 10984 SHALL ignore any value provided in this field.
- 10985 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- 10986 • Set *pNonce* to hold the nonce data returned from **C_WrapKey**. If *ulNonceFixedBits* is not zero,
- 10987 then the most significant bits of *pNonce* contain the fixed nonce. If *nonceGenerator* is set to
- 10988 **CKG_NO_GENERATE**, *pNonce* is an input parameter with the full nonce.
- 10989 • Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.
- 10990 • Set the MAC length *ulMACLen* in the parameter block.
- 10991 • Call **C_WrapKey** for **CKM_AES_CCM** mechanism with parameters, wrapping key *wK*, key to be
- 10992 wrapped *mK*, obtaining the final wrapped key and the MAC. The total length of data processed
- 10993 must be *ulDataLen*. The output length will be *ulDataLen* + *ulMACLen*.

10994 UnwrapKey:

- 10995 • Set the message/data length *ulDataLen* to Zero in the parameter block. The token will get the
- 10996 ciphertext length from *ulWrappedKeyLen*.
- 10997 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- 10998 • The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.

- 10999 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
- 11000 *ulAADLen* is 0.
- 11001 • Set the MAC length *ulMACLen* in the parameter block.
- 11002 • Call **C_UnwrapKey** for **CKM_AES_CCM** mechanism with parameters, unwrapping key *wK*,
- 11003 template, and wrapped key including the appended MAC, obtaining a new key handle.
- 11004 WrapKeyAuthenticated:
- 11005 • Set the message/data length *ulDataLen* in the parameter block to zero, the length will be defined
- 11006 by the token based on the key being wrapped.
- 11007 • Set the nonce length *ulNonceLen*.
- 11008 • Set *pNonce* to hold the nonce data returned from **C_WrapKeyAuthenticated**. If *ulNonceFixedBits*
- 11009 is not zero, then the most significant bits of *pNonce* contain the fixed nonce. If *nonceGenerator* is
- 11010 set to **CKG_NO_GENERATE**, *pNonce* is an input parameter with the full nonce.
- 11011 • Set the *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block.
- 11012 • Set the MAC length *ulMACLen* in the parameter block.
- 11013 • Set *pMAC* to hold the MAC data returned from **C_WrapKeyAuthenticated**.
- 11014 • Call **C_WrapKeyAuthenticated** for **CKM_AES_CCM** mechanism with wrapping key *wK*, the key
- 11015 to be wrapped *mK*, and parameter block.
- 11016 • The MAC is returned in *pMac* of the CK_CCM_MESSAGE_PARAMS structure.
- 11017 UnwrapKeyAuthenticated:
- 11018 • Set the message/data length *ulDataLen* to Zero in the parameter block.
- 11019 • Set the nonce length *ulNonceLen* and the nonce data *pNonce* in the parameter block.
- 11020 • The *ulNonceFixedBits* and *nonceGenerator* fields in the parameter block are ignored.
- 11021 • Set the MAC length *ulMACLen* in the parameter block.
- 11022 • Set the MAC data *pMAC* in the parameter block.
- 11023 • Call **C_UnwrapKeyAuthenticated** for **CKM_AES_CCM** mechanism with unwrapping key *wK*,
- 11024 wrapped key *mK*, parameter block and template, obtaining a new key handle.
- 11025
- 11026 In *pNonce* the least significant bit of the nonce is the rightmost bit. *ulNonceLen* is the length of the nonce
- 11027 in bytes.
- 11028 On **C_WrapKeyAuthenticated** the meaning of *nonceGenerator* is as follows:
- 11029 **CKG_NO_GENERATE** means the nonce is passed in on WrapKeyAuthenticated and no internal nonce
- 11030 generation is done.
- 11031 **CKG_GENERATE** means that the non-fixed portion of the nonce is generated by the module internally.
- 11032 The generation method is not defined.
- 11033 **CKG_GENERATE_COUNTER** means that the non-fixed portion of the nonce is generated by the module
- 11034 internally by use of an incrementing counter, the initial IV counter is zero.
- 11035 **CKG_GENERATE_COUNTER_XOR** means that the non-fixed portion of the IV is xored with a counter.
- 11036 The value of the non-fixed portion passed must not vary from call to call. Like
- 11037 **CKG_GENERATE_COUNTER**, the counter starts at zero.
- 11038 **CKG_GENERATE_RANDOM** means that the non-fixed portion of the nonce is generated by the module
- 11039 internally using a PRNG. In any case the entire nonce, including the fixed portion, is returned in *pNonce*.
- 11040 Modules must implement **CKG_GENERATE**. Modules may also reject *ulNonceFixedBits* values which
- 11041 are too large. Zero is always an acceptable value for *ulNonceFixedBits*.

11042 In WrapKey and UnwrapKey the MAC is appended to the ciphertext and the least significant byte of the
11043 MAC is the rightmost byte and the MAC bytes are the rightmost *ulMACLen* bytes. In
11044 **C_WrapKeyAuthenticated** the MAC is returned in the *pMAC* field of CK_CCM_MESSAGE_PARAMS. In
11045 **C_UnwrapKeyAuthenticated** the MAC is provided by the *pMAC* field of
11046 CK_CCM_MESSAGE_PARAMS.

11047 **6.13.6 AES-GMAC**

11048 AES-GMAC, denoted **CKM_AES_GMAC**, is a mechanism for single and multiple-part signatures and
11049 verification. It is described in NIST Special Publication 800-38D [GMAC]. GMAC is a special case of GCM
11050 that authenticates only the Additional Authenticated Data (AAD) part of the GCM mechanism parameters.
11051 When GMAC is used with **C_Sign** or **C_Verify**, *pData* points to the AAD. GMAC does not use plaintext or
11052 ciphertext.

11053 The signature produced by GMAC, also referred to as a Tag, the tag's length is determined by the
11054 CK_GCM_PARAMS field *ulTagBits*.

11055 The IV length is determined by the CK_GCM_PARAMS field *ulIvLen*.

11056 Constraints on key types and the length of data are summarized in the following table:

11057 *Table 119, AES-GMAC: Key And Data Length*

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

11058 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11059 specify the supported range of AES key sizes, in bytes.

11060 **6.13.7 AES GCM and CCM Mechanism parameters**

11061 **♦ CK_GENERATOR_FUNCTION**

11062 Functions to generate unique IVs and nonces.

11063 `typedef CK_ULONG CK_GENERATOR_FUNCTION;`

11064 **♦ CK_GCM_PARAMS; CK_GCM_PARAMS_PTR**

11065 CK_GCM_PARAMS is a structure that provides the parameters to the **CKM_AES_GCM** mechanism
11066 when used for Encrypt or Decrypt. It is defined as follows:

```
11067 typedef struct CK_GCM_PARAMS {  
11068     CK_BYTE_PTR    pIv;  
11069     CK_ULONG       ulIvLen;  
11070     CK_ULONG       ulIvBits;  
11071     CK_BYTE_PTR    pAAD;  
11072     CK_ULONG       ulAADLen;  
11073     CK_ULONG       ulTagBits;  
11074 } CK_GCM_PARAMS;
```

11075
11076 The fields of the structure have the following meanings:

11077 `plv` pointer to initialization vector

11078	ulIvLen	length of initialization vector in bytes. The length of the initialization
11079		vector can be any number between 1 and $(2^{32}) - 1$. 96-bit (12 byte)
11080		IV values can be processed more efficiently, so that length is
11081		recommended for situations in which efficiency is critical.
11082	ulIvBits	length of initialization vector in bits. Do not use ulIvBits to specify the
11083		length of the initialization vector, but ulIvLen instead.
11084	pAAD	pointer to additional authentication data. This data is authenticated
11085		but not encrypted.
11086	ulAADLen	length of pAAD in bytes. The length of the AAD can be any number
11087		between 0 and $(2^{32}) - 1$.
11088	ulTagBits	length of authentication tag (output following ciphertext) in bits. Can
11089		be any value between 0 and 128.

11090 **CK_GCM_PARAMS_PTR** is a pointer to a **CK_GCM_PARAMS**.

11091 ♦ **CK_GCM_MESSAGE_PARAMS;**
 11092 **CK_GCM_MESSAGE_PARAMS_PTR**

11093 **CK_GCM_MESSAGE_PARAMS** is a structure that provides the parameters to the **CKM_AES_GCM**
 11094 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```

11095     typedef struct CK_GCM_MESSAGE_PARAMS {
11096         CK_BYTE_PTR    pIv;
11097         CK_ULONG       ulIvLen;
11098         CK_ULONG       ulIvFixedBits;
11099         CK_GENERATOR_FUNCTION  ivGenerator;
11100         CK_BYTE_PTR    pTag;
11101         CK_ULONG       ulTagBits;
11102     } CK_GCM_MESSAGE_PARAMS;
  
```

11103
 11104 The fields of the structure have the following meanings:

11105	pIv	pointer to initialization vector
11106	ulIvLen	length of initialization vector in bytes. The length of the initialization
11107		vector can be any number between 1 and $(2^{32}) - 1$. 96-bit (12 byte)
11108		IV values can be processed more efficiently, so that length is
11109		recommended for situations in which efficiency is critical.
11110	ulIvFixedBits	number of bits of the original IV to preserve when generating an
11111		new IV. These bits are counted from the Most significant bits (to the
11112		right).
11113	ivGenerator	Function used to generate a new IV. Each IV must be unique for a
11114		given session.
11115	pTag	location of the authentication tag which is returned on
11116		MessageEncrypt, and provided on MessageDecrypt.
11117	ulTagBits	length of authentication tag in bits. Can be any value between 0 and
11118		128.

11119 **CK_GCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_GCM_MESSAGE_PARAMS**.

11120 ♦ **CK_GCM_WRAP_PARAMS; CK_GCM_WRAP_PARAMS_PTR**

11121 **CK_GCM_WRAP_PARAMS** is a structure that provides the parameters to the **CKM_AES_GCM**
 11122 mechanism when used for WrapKey to return a token generated IV for input into UnwrapKey. It is defined
 11123 as follows:

```
11124     typedef struct CK_GCM_WRAP_PARAMS {
11125         CK_BYTE_PTR    pIv;
11126         CK_ULONG       ulIvLen;
11127         CK_ULONG       ulIvFixedBits;
11128         CK_GENERATOR_FUNCTION    ivGenerator;
11129         CK_BYTE_PTR    pAAD;
11130         CK_ULONG       ulAADLen;
11131         CK_ULONG       ulTagBits;
11132     } CK_GCM_WRAP_PARAMS;
```

11133

11134 The fields of the structure have the following meanings:

11135	pIv	pointer to initialization vector
11136	ulIvLen	length of initialization vector in bytes. The length of the initialization
11137		vector can be any number between 1 and (2 ³²) - 1. 96-bit (12 byte)
11138		IV values can be processed more efficiently, so that length is
11139		recommended for situations in which efficiency is critical.
11140	ulIvFixedBits	number of bits of the original IV to preserve when generating an
11141		new IV. These bits are counted from the Most significant bits (to the
11142		right).
11143	ivGenerator	Function used to generate a new IV. Each IV must be unique for a
11144		given session.
11145	pAAD	pointer to additional authentication data. This data is authenticated
11146		but not encrypted.
11147	ulAADLen	length of pAAD in bytes. The length of the AAD can be any number
11148		between 0 and (2 ³²) - 1.
11149	ulTagBits	length of authentication tag in bits. Can be any value between 0 and
11150		128.

11151 **CK_GCM_WRAP_PARAMS_PTR** is a pointer to a **CK_GCM_WRAP_PARAMS**.

11152 ♦ **CK_CCM_PARAMS; CK_CCM_PARAMS_PTR**

11153 **CK_CCM_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM** mechanism
 11154 when used for Encrypt or Decrypt. It is defined as follows:

```
11155     typedef struct CK_CCM_PARAMS {
11156         CK_ULONG       ulDataLen; /*plaintext or ciphertext*/
11157         CK_BYTE_PTR    pNonce;
11158         CK_ULONG       ulNonceLen;
11159         CK_BYTE_PTR    pAAD;
11160         CK_ULONG       ulAADLen;
11161         CK_ULONG       ulMACLen;
11162     } CK_CCM_PARAMS;
```

11163

11164 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
11165 length ($2 \leq L \leq 8$):

11166	ulDataLen	length of the data where $0 \leq \text{ulDataLen} < 2^{(8L)}$.
11167	pNonce	the nonce.
11168	ulNonceLen	length of pNonce in bytes where $7 \leq \text{ulNonceLen} \leq 13$.
11169	pAAD	Additional authentication data. This data is authenticated but not
11170		encrypted.
11171	ulAADLen	length of pAAD in bytes where $0 \leq \text{ulAADLen} \leq (2^{32}) - 1$.
11172	ulMACLen	length of the MAC (output following ciphertext) in bytes. Valid values
11173		are 4, 6, 8, 10, 12, 14, and 16.

11174 **CK_CCM_PARAMS_PTR** is a pointer to a **CK_CCM_PARAMS**.

11175 ♦ **CK_CCM_MESSAGE_PARAMS;**
11176 **CK_CCM_MESSAGE_PARAMS_PTR**

11177 **CK_CCM_MESSAGE_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM**
11178 mechanism when used for MessageEncrypt or MessageDecrypt. It is defined as follows:

```
11179     typedef struct CK_CCM_MESSAGE_PARAMS {  
11180         CK_ULONG      ulDataLen; /*plaintext or ciphertext*/  
11181         CK_BYTE_PTR   pNonce;  
11182         CK_ULONG      ulNonceLen;  
11183         CK_ULONG      ulNonceFixedBits;  
11184         CK_GENERATOR_FUNCTION  nonceGenerator;  
11185         CK_BYTE_PTR   pMAC;  
11186         CK_ULONG      ulMACLen;  
11187     } CK_CCM_MESSAGE_PARAMS;
```

11188

11189 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
11190 length ($2 \leq L \leq 8$):

11191	ulDataLen	length of the data where $0 \leq \text{ulDataLen} < 2^{(8L)}$.
11192	pNonce	the nonce.
11193	ulNonceLen	length of pNonce in bytes where $7 \leq \text{ulNonceLen} \leq 13$.
11194	ulNonceFixedBits	number of bits of the original nonce to preserve when generating a
11195		new nonce. These bits are counted from the Most significant bits (to
11196		the right).
11197	nonceGenerator	Function used to generate a new nonce. Each nonce must be
11198		unique for a given session.
11199	pMAC	location of the CCM MAC returned on MessageEncrypt, provided on
11200		MessageDecrypt
11201	ulMACLen	length of the MAC (output following ciphertext) in bytes. Valid values
11202		are 4, 6, 8, 10, 12, 14, and 16.

11203 **CK_CCM_MESSAGE_PARAMS_PTR** is a pointer to a **CK_CCM_MESSAGE_PARAMS**.

11204 ♦ **CK_CCM_WRAP_PARAMS; CK_CCM_WAP_PARAMS_PTR**

11205 **CK_CCM_WRAP_PARAMS** is a structure that provides the parameters to the **CKM_AES_CCM**
11206 mechanism when used for WrapKey only to provide a token generated nonce and the number of bits to
11207 preserve. It is defined as follows:

```
11208     typedef struct CK_CCM_WRAP_PARAMS {  
11209         CK_ULONG      ulDataLen; /*wrappedkey data*/  
11210         CK_BYTE_PTR   pNonce;  
11211         CK_ULONG      ulNonceLen;  
11212         CK_ULONG      ulNonceFixedBits;  
11213         CK_GENERATOR_FUNCTION   nonceGenerator;  
11214         CK_BYTE_PTR   pAAD;  
11215         CK_ULONG      ulAADLen;  
11216         CK_ULONG      ulMACLen;  
11217     } CK_CCM_WRAP_PARAMS;  
11218  
11219
```

11220 The fields of the structure have the following meanings, where L is the size in bytes of the data length's
11221 length ($2 \leq L \leq 8$):

- 11222 ulDataLen length of the data where $0 \leq \text{ulDataLen} < 2^{(8L)}$.
- 11223 pNonce the nonce.
- 11224 ulNonceLen length of pNonce in bytes where $7 \leq \text{ulNonceLen} \leq 13$.
- 11225 ulNonceFixedBits number of bits of the original nonce to preserve when generating a
11226 new nonce. These bits are counted from the most significant bits (to
11227 the right).
- 11228 nonceGenerator Function used to generate a new nonce. Each nonce must be
11229 unique for a given session.
- 11230 pAAD Additional authentication data. This data is authenticated but not
11231 wrapped.
- 11232 ulAADLen length of pAAD in bytes where $0 \leq \text{ulAADLen} \leq (2^{32}) - 1$.
- 11233 ulMACLen length of the MAC (output following ciphertext) in bytes. Valid values
11234 are 4, 6, 8, 10, 12, 14, and 16.

11235 **CK_CCM_WRAP_PARAMS_PTR** is a pointer to a **CK_CCM_WRAP_PARAMS**.

11236 **6.14 AES CMAC**

11237 *Table 120, Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_AES_CMAC_GENERAL		✓						
CKM_AES_CMAC		✓						

6.14.1 Definitions

Mechanisms:

CKM_AES_CMACE_GENERAL

CKM_AES_CMACE

6.14.2 Mechanism parameters

CKM_AES_CMACE_GENERAL uses the existing CK_MAC_GENERAL_PARAMS structure.

CKM_AES_CMACE does not use a mechanism parameter.

6.14.3 General-length AES-CMAC

General-length AES-CMAC, denoted CKM_AES_CMACE_GENERAL, 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 121, 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

[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_CMACE, 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 122, 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)

[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 123, Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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 124, 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}	CK_ULONG	Length in bytes of key value

Refer to Table 13 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 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:

11298 Table 125, 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

11299

11300 6.16 AES Key Wrap

11301 Table 126, AES Key Wrap Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_AES_KEY_WRAP	✓					✓		
CKM_AES_KEY_WRAP_PAD	✓					✓		
CKM_AES_KEY_WRAP_KWP	✓					✓		
CKM_AES_KEY_WRAP_PKCS7	✓					✓		

11302 6.16.1 Definitions

11303 Mechanisms:

- 11304 CKM_AES_KEY_WRAP
- 11305 CKM_AES_KEY_WRAP_PAD
- 11306 CKM_AES_KEY_WRAP_KWP
- 11307 CKM_AES_KEY_WRAP_PKCS7

11308 6.16.2 AES Key Wrap Mechanism parameters

11309 The mechanisms will accept an optional mechanism parameter as the Initialization vector which, if
11310 present, must be a fixed size array of 8 bytes for **CKM_AES_KEY_WRAP** and
11311 **CKM_AES_KEY_WRAP_PKCS7**, resp. 4 bytes for **CKM_AES_KEY_WRAP_KWP**; and, if NULL, will
11312 use the default initial value defined in section 4.3 resp. 6.2 / 6.3 of [AES KEYWRAP].

11313 The type of this parameter is CK_BYTE_PTR and the pointer points to the array of bytes to be used as
11314 the initial value. The length shall be either 0 and the pointer NULL; or 8 for **CKM_AES_KEY_WRAP** and
11315 **CKM_AES_KEY_WRAP_PKCS7**, resp. 4 for **CKM_AES_KEY_WRAP_KWP**, and the pointer non-NULL.

11316 6.16.3 AES Key Wrap

11317 The mechanisms support only single-part operations, i.e. single part wrapping and unwrapping, and
11318 single-part encryption and decryption.

11319 ♦ CKM_AES_KEY_WRAP

11320 The **CKM_AES_KEY_WRAP** mechanism can wrap a key of any length. A secret key whose length is not
11321 a multiple of the AES Key Wrap semiblock size (8 bytes) will be zero padded to fit. Semiblock size is
11322 defined in section 5.2 of [AES KEYWRAP]. A private key will be encoded as defined in section 6.7; the
11323 encoded private key will be zero padded to fit if necessary.

11324

11325 The **CKM_AES_KEY_WRAP** mechanism can only encrypt a block of data whose size is an exact

11326 multiple of the AES Key Wrap algorithm semiblock size.

11327

11328 For unwrapping, the mechanism decrypts the wrapped key. In case of a secret key, it truncates the result

11329 according to the **CKA_KEY_TYPE** attribute of the template and, if it has one and the key type supports it,

11330 the **CKA_VALUE_LEN** attribute of the template. The length specified in the template must not be less

11331 than n-7 bytes, where n is the length of the wrapped key. In case of a private key, the mechanism parses

11332 the encoding as defined in section 6.7 and ignores trailing zero bytes.

11333 ♦ **CKM_AES_KEY_WRAP_PAD**

11334 The **CKM_AES_KEY_WRAP_PAD** mechanism is deprecated. **CKM_AES_KEY_WRAP_KWP** resp.

11335 **CKM_AES_KEY_WRAP_PKCS7** shall be used instead.

11336 ♦ **CKM_AES_KEY_WRAP_KWP**

11337 The **CKM_AES_KEY_WRAP_KWP** mechanism can wrap a key or encrypt block of data of any length.

11338 The input is zero-padded and wrapped / encrypted as defined in section 6.3 of [AES KEYWRAP], which

11339 produces same results as RFC 5649.

11340 ♦ **CKM_AES_KEY_WRAP_PKCS7**

11341 The **CKM_AES_KEY_WRAP_PKCS7** mechanism can wrap a key or encrypt a block of data of any

11342 length. It does the padding detailed in [PKCS #7] of inputs (keys or data blocks) up to a semiblock size to

11343 make it an exact multiple of AES Key Wrap algorithm semiblock size (8bytes), always producing wrapped

11344 output that is larger than the input key/data to be wrapped. This padding is done by the token before

11345 being passed to the AES key wrap algorithm, which then wraps / encrypts the padded block of data as

11346 defined in section 6.2 of [AES KEYWRAP].

11347 **6.17 Key derivation by data encryption – DES & AES**

11348 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.

11349 They are for use with the **C_DeriveKey** function.

11350 *Table 127, Key derivation by data encryption Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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							✓	

11351 **6.17.1 Definitions**

11352 Mechanisms:

11353 CKM_DES_ECB_ENCRYPT_DATA

```
11354     CKM_DES_CBC_ENCRYPT_DATA
11355     CKM_DES3_ECB_ENCRYPT_DATA
11356     CKM_DES3_CBC_ENCRYPT_DATA
11357     CKM_AES_ECB_ENCRYPT_DATA
11358     CKM_AES_CBC_ENCRYPT_DATA
11359
11360     typedef struct CK_DES_CBC_ENCRYPT_DATA_PARAMS {
11361         CK_BYTE      iv[8];
11362         CK_BYTE_PTR   pData;
11363         CK_ULONG      length;
11364     } CK_DES_CBC_ENCRYPT_DATA_PARAMS;
11365
11366     typedef CK_DES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
11367         CK_DES_CBC_ENCRYPT_DATA_PARAMS_PTR;
11368
11369     typedef struct CK_AES_CBC_ENCRYPT_DATA_PARAMS {
11370         CK_BYTE      iv[16];
11371         CK_BYTE_PTR   pData;
11372         CK_ULONG      length;
11373     } CK_AES_CBC_ENCRYPT_DATA_PARAMS;
11374
11375     typedef CK_AES_CBC_ENCRYPT_DATA_PARAMS CK_PTR
11376         CK_AES_CBC_ENCRYPT_DATA_PARAMS_PTR;
```

11377 **6.17.2 Mechanism Parameters**

11378 Uses CK_KEY_DERIVATION_STRING_DATA as defined in section 6.43.2

11379 *Table 128, 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.

11380 **6.17.3 Mechanism Description**

11381 The mechanisms will function by performing the encryption over the data provided using the base key.
11382 The resulting ciphertext shall be used to create the key value of the resulting key. If not all the ciphertext
11383 is used then the part discarded will be from the trailing end (least significant bytes) of the ciphertext data.
11384 The derived key shall be defined by the attribute template supplied but constrained by the length of
11385 ciphertext available for the key value and other normal PKCS11 derivation constraints.

11386 Attribute template handling, attribute defaulting and key value preparation will operate as per the SHA-1
11387 Key Derivation mechanism in section 6.20.5.
11388 If the data is too short to make the requested key then the mechanism returns
11389 **CKR_DATA_LEN_RANGE**.

11390 **6.18 Double and Triple-length DES**

11391 *Table 129, Double and Triple-Length DES Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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		✓						

11392 **6.18.1 Definitions**

11393 This section defines the key type “**CKK_DES2**” and “**CKK_DES3**” for type CK_KEY_TYPE as used in the
11394 **CKA_KEY_TYPE** attribute of key objects.

11395 Mechanisms:

- 11396 CKM_DES2_KEY_GEN
- 11397 CKM_DES3_KEY_GEN
- 11398 CKM_DES3_ECB
- 11399 CKM_DES3_CBC
- 11400 CKM_DES3_MAC
- 11401 CKM_DES3_MAC_GENERAL
- 11402 CKM_DES3_CBC_PAD

11403 **6.18.2 DES2 secret key objects**

11404 DES2 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES2**) hold double-length
11405 DES keys. The following table defines the DES2 secret key object attributes, in addition to the common
11406 attributes defined for this object class:

11407 *Table 130, DES2 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 16 bytes long)

11408 ¹ Refer to Table 13 for footnotes

11409 DES2 keys must always have their parity bits properly set as described in [FIPS PUB 46-3] (*i.e.*, each of
11410 the DES keys comprising a DES2 key must have its parity bits properly set). Attempting to create or
11411 unwrap a DES2 key with incorrect parity will return an error.

11412 The following is a sample template for creating a double-length DES secret key object:

```
11413 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
11414 CK_KEY_TYPE keyType = CKK_DES2;
11415 CK_UTF8CHAR label[] = "A DES2 secret key object";
11416 CK_BYTE value[16] = {...};
11417 CK_BBOOL true = CK_TRUE;
11418 CK_ATTRIBUTE template[] = {
11419     {CKA_CLASS, &class, sizeof(class)},
11420     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
11421     {CKA_TOKEN, &true, sizeof(true)},
11422     {CKA_LABEL, label, sizeof(label)-1},
11423     {CKA_ENCRYPT, &true, sizeof(true)},
11424     {CKA_VALUE, value, sizeof(value)}
11425 };
11426
```

11427 **CKA_CHECK_VALUE:** The value of this attribute is derived from the key object by taking the first three
11428 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
11429 the key type of the secret key object.

11430 **6.18.3 DES3 secret key objects**

11431 DES3 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_DES3**) hold triple-length DES
11432 keys. The following table defines the DES3 secret key object attributes, in addition to the common
11433 attributes defined for this object class:

11434 *Table 131, DES3 Secret Key Object Attributes*

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 24 bytes long)

11435 Refer to Table 13 for footnotes

11436 DES3 keys must always have their parity bits properly set as described in [FIPS PUB 46-3] (*i.e.*, each of
11437 the DES keys comprising a DES3 key must have its parity bits properly set). Attempting to create or
11438 unwrap a DES3 key with incorrect parity will return an error.

11439 The following is a sample template for creating a triple-length DES secret key object:

```
11440 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
11441 CK_KEY_TYPE keyType = CKK_DES3;
11442 CK_UTF8CHAR label[] = "A DES3 secret key object";
11443 CK_BYTE value[24] = {...};
11444 CK_BBOOL true = CK_TRUE;
11445 CK_ATTRIBUTE template[] = {
11446     {CKA_CLASS, &class, sizeof(class)},
11447     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
11448     {CKA_TOKEN, &true, sizeof(true)},
11449     {CKA_LABEL, label, sizeof(label)-1},
11450     {CKA_ENCRYPT, &true, sizeof(true)},
11451     {CKA_VALUE, value, sizeof(value)}
11452 };
11453
```

11454 **CKA_CHECK_VALUE:** The value of this attribute is derived from the key object by taking the first three
11455 bytes of the ECB encryption of a single block of null (0x00) bytes, using the default cipher associated with
11456 the key type of the secret key object.

6.18.4 Double-length DES key generation

The double-length DES key generation mechanism, denoted **CKM_DES2_KEY_GEN**, is a key generation mechanism for double-length DES keys. The DES keys making up a double-length DES key both have their parity bits set properly, as specified in [FIPS PUB 46-3].

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 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 TRIPLE DATA ENCRYPTION ALGORITHM (TDEA) A Cipher Block Chaining Mode of Operation (TCBC) as defined in [FIPS PUB 46-3]. 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 ciphertext 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 132, DES and Triple Length DES in OFB Mode Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_DES_OFB64	✓							
CKM_DES_OFB8	✓							
CKM_DES_CFB64	✓							
CKM_DES_CFB8	✓							

11490 Cipher DES has a output feedback mode, DES-OFB, denoted **CKM_DES_OFB8** and
11491 **CKM_DES_OFB64**. It is a mechanism for single and multiple-part encryption and decryption with DES.
11492 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
11493 the block size.
11494 Constraints on key types and the length of data are summarized in the following table:
11495 *Table 133, 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

11496 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

11497 **6.18.8 DES and Triple length DES in CFB Mode**

11498 Cipher DES has a cipher feedback mode, DES-CFB, denoted **CKM_DES_CFB8** and **CKM_DES_CFB64**.
11499 It is a mechanism for single and multiple-part encryption and decryption with DES.
11500 It has a parameter, an initialization vector for this mode. The initialization vector has the same length as
11501 the block size.
11502 Constraints on key types and the length of data are summarized in the following table:
11503 *Table 134, 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

11504 For this mechanism the **CK_MECHANISM_INFO** structure is as specified for CBC mode.

11505 **6.19 Double and Triple-length DES CMAC**

11506 *Table 135, Double and Triple-length DES CMAC Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_DES3_CMACH_GENERAL		✓						
CKM_DES3_CMACH		✓						

11507 **6.19.1 Definitions**

11508 Mechanisms:

11509 CKM_DES3_CMAC_GENERAL
11510 CKM_DES3_CMAC

11511 **6.19.2 Mechanism parameters**

11512 **CKM_DES3_CMAC_GENERAL** uses the existing **CK_MAC_GENERAL_PARAMS** structure.
11513 **CKM_DES3_CMAC** does not use a mechanism parameter.

11514 **6.19.3 General-length DES3-MAC**

11515 General-length DES3-CMAC, denoted **CKM_DES3_CMAC_GENERAL**, is a mechanism for single- and
11516 multiple-part signatures and verification with DES3 or DES2 keys, based on [NIST sp800-38b].

11517 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
11518 desired from the mechanism.

11519 The output bytes from this mechanism are taken from the start of the final DES3 cipher block produced in
11520 the MACing process.

11521 Constraints on key types and the length of data are summarized in the following table:

11522 *Table 136, 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

11523 Reference [NIST sp800-38b] recommends that the output MAC is not truncated to less than 64 bits
11524 (which means using the entire block for DES). The MAC length must be specified before the
11525 communication starts, and must not be changed during the lifetime of the key. It is the caller's
11526 responsibility to follow these rules.

11527 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11528 are not used

11529 **6.19.4 DES3-CMAC**

11530 DES3-CMAC, denoted **CKM_DES3_CMAC**, is a special case of the general-length DES3-CMAC
11531 mechanism. DES3-MAC always produces and verifies MACs that are a full block size in length, since the
11532 DES3 block length is the minimum output length recommended by [NIST sp800-38b].

11533 Constraints on key types and the length of data are summarized in the following table:

11534 *Table 137, 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)

11535 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11536 are not used.

11537 **6.20 SHA-1**

11538 *Table 138, SHA-1 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA_1				✓				
CKM_SHA_1_HMAC_GENERAL		✓						
CKM_SHA_1_HMAC		✓						
CKM_SHA1_KEY_DERIVATION							✓	
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 180-4].

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 139, 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 140, 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.

11601 The mechanism generates SHA-1-HMAC keys with a particular length in bytes, as specified in the
11602 **CKA_VALUE_LEN** attribute of the template for the key.

11603 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11604 key. Other attributes supported by the SHA-1-HMAC key type (specifically, the flags indicating which
11605 functions the key supports) may be specified in the template for the key, or else are assigned default
11606 initial values.

11607 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11608 specify the supported range of **CKM_SHA_1_HMAC** key sizes, in bytes.

11609 **6.21 SHA-224**

11610 *Table 141, SHA-224 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA224				✓				
CKM_SHA224_HMAC		✓						
CKM_SHA224_HMAC_GENERAL		✓						
CKM_SHA224_KEY_DERIVATION							✓	
CKM_SHA224_KEY_GEN					✓			

11611 **6.21.1 Definitions**

11612 This section defines the key type “**CKK_SHA224_HMAC**” for type CK_KEY_TYPE as used in the
11613 **CKA_KEY_TYPE** attribute of key objects.

11614 Mechanisms:

- 11615 CKM_SHA224
- 11616 CKM_SHA224_HMAC
- 11617 CKM_SHA224_HMAC_GENERAL
- 11618 CKM_SHA224_KEY_DERIVATION
- 11619 CKM_SHA224_KEY_GEN

11620 **6.21.2 SHA-224 digest**

11621 The SHA-224 mechanism, denoted **CKM_SHA224**, is a mechanism for message digesting, following the
11622 Secure Hash Algorithm with a 224-bit message digest defined in [FIPS PUB 180-4].

11623 It does not have a parameter.

11624 Constraints on the length of input and output data are summarized in the following table. For single-part
11625 digesting, the data and the digest may begin at the same location in memory.

11626 *Table 142, SHA-224: Data Length*

Function	Input length	Digest length
C_Digest	any	28

11627 **6.21.3 General-length SHA-224-HMAC**

11628 The general-length SHA-224-HMAC mechanism, denoted **CKM_SHA224_HMAC_GENERAL**, is the
11629 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 143, 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 144, SHA-256 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_SHA256				✓				

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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 **CKA_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 180-4].

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 145, 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.

11688 Table 146, 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

11689 6.22.4 SHA-256-HMAC

11690 The SHA-256-HMAC mechanism, denoted **CKM_SHA256_HMAC**, is a special case of the general-length
11691 SHA-256-HMAC mechanism in section 6.22.3.

11692 It has no parameter, and always produces an output of length 32.

11693 6.22.5 SHA-256 key derivation

11694 SHA-256 key derivation, denoted **CKM_SHA256_KEY_DERIVATION**, is the same as the SHA-1 key
11695 derivation mechanism in section 6.20.5, except that it uses the SHA-256 hash function and the relevant
11696 length is 32 bytes.

11697 6.22.6 SHA-256 HMAC key generation

11698 The SHA-256-HMAC key generation mechanism, denoted **CKM_SHA256_KEY_GEN**, is a key
11699 generation mechanism for NIST's SHA256-HMAC.

11700 It does not have a parameter.

11701 The mechanism generates SHA256-HMAC keys with a particular length in bytes, as specified in the
11702 **CKA_VALUE_LEN** attribute of the template for the key.

11703 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11704 key. Other attributes supported by the SHA256-HMAC key type (specifically, the flags indicating which
11705 functions the key supports) may be specified in the template for the key, or else are assigned default
11706 initial values.

11707 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11708 specify the supported range of **CKM_SHA256_HMAC** key sizes, in bytes.

11709 6.23 SHA-384

11710 Table 147, SHA-384 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA384				✓				
CKM_SHA384_HMAC_GENERAL		✓						
CKM_SHA384_HMAC		✓						
CKM_SHA384_KEY_DERIVATION							✓	
CKM_SHA384_KEY_GEN					✓			

11711 6.23.1 Definitions

11712 This section defines the key type "**CKK_SHA384_HMAC**" for type CK_KEY_TYPE as used in the
11713 **CKA_KEY_TYPE** attribute of key objects.

- 11714 CKM_SHA384
- 11715 CKM_SHA384_HMAC
- 11716 CKM_SHA384_HMAC_GENERAL
- 11717 CKM_SHA384_KEY_DERIVATION
- 11718 CKM_SHA384_KEY_GEN

11719 **6.23.2 SHA-384 digest**

- 11720 The SHA-384 mechanism, denoted **CKM_SHA384**, is a mechanism for message digesting, following the
- 11721 Secure Hash Algorithm with a 384-bit message digest defined in [FIPS PUB 180-4].
- 11722 It does not have a parameter.
- 11723 Constraints on the length of input and output data are summarized in the following table. For single-part
- 11724 digesting, the data and the digest may begin at the same location in memory.
- 11725 *Table 148, SHA-384: Data Length*

Function	Input length	Digest length
C_Digest	any	48

11726 **6.23.3 General-length SHA-384-HMAC**

- 11727 The general-length SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC_GENERAL**, is the
- 11728 same as the general-length SHA-1-HMAC mechanism in section 6.20.3, except that it uses the HMAC
- 11729 construction based on the SHA-384 hash function and length of the output should be in the range 1-48.
- 11730 The keys it uses are generic secret keys and **CKK_SHA384_HMAC**. FIPS-198 compliant tokens may
- 11731 require the key length to be at least 24 bytes; that is, half the size of the SHA-384 hash output.
- 11732 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
- 11733 output. This length should be in the range 0-48 (the output size of SHA-384 is 48 bytes). FIPS-198
- 11734 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).
- 11735 Signatures (MACs) produced by this mechanism will be taken from the start of the full 48-byte HMAC
- 11736 output.
- 11737 *Table 149, 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

11738

11739 **6.23.4 SHA-384-HMAC**

- 11740 The SHA-384-HMAC mechanism, denoted **CKM_SHA384_HMAC**, is a special case of the general-length
- 11741 SHA-384-HMAC mechanism.
- 11742 It has no parameter, and always produces an output of length 48.

11743 **6.23.5 SHA-384 key derivation**

- 11744 SHA-384 key derivation, denoted **CKM_SHA384_KEY_DERIVATION**, is the same as the SHA-1 key
- 11745 derivation mechanism in section 6.20.5, except that it uses the SHA-384 hash function and the relevant
- 11746 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 150, SHA-512 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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 180-4].

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.

11776 *Table 151, SHA-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

11777 **6.24.3 General-length SHA-512-HMAC**

11778 The general-length SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC_GENERAL**, is the
11779 same as the general-length SHA-1-HMAC mechanism in section 6.20.3, except that it uses the HMAC
11780 construction based on the SHA-512 hash function and length of the output should be in the range 1-64.

11781 The keys it uses are generic secret keys and **CKK_SHA512_HMAC**. FIPS-198 compliant tokens may
11782 require the key length to be at least 32 bytes; that is, half the size of the SHA-512 hash output.

11783 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
11784 output. This length should be in the range 0-64 (the output size of SHA-512 is 64 bytes). FIPS-198
11785 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length).
11786 Signatures (MACs) produced by this mechanism will be taken from the start of the full 64-byte HMAC
11787 output.

11788 *Table 152, 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

11789

11790 **6.24.4 SHA-512-HMAC**

11791 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_HMAC**, is a special case of the general-length
11792 SHA-512-HMAC mechanism.

11793 It has no parameter, and always produces an output of length 64.

11794 **6.24.5 SHA-512 key derivation**

11795 SHA-512 key derivation, denoted **CKM_SHA512_KEY_DERIVATION**, is the same as the SHA-1 key
11796 derivation mechanism in section 6.20.5, except that it uses the SHA-512 hash function and the relevant
11797 length is 64 bytes.

11798 **6.24.6 SHA-512 HMAC key generation**

11799 The SHA-512-HMAC key generation mechanism, denoted **CKM_SHA512_KEY_GEN**, is a key
11800 generation mechanism for NIST's SHA512-HMAC.

11801 It does not have a parameter.

11802 The mechanism generates SHA512-HMAC keys with a particular length in bytes, as specified in the
11803 **CKA_VALUE_LEN** attribute of the template for the key.

11804 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11805 key. Other attributes supported by the SHA512-HMAC key type (specifically, the flags indicating which
11806 functions the key supports) may be specified in the template for the key, or else are assigned default
11807 initial values.

11808 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11809 specify the supported range of **CKM_SHA512_HMAC** key sizes, in bytes.

6.25 SHA-512/224

Table 153, SHA-512/224 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
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 154, 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.

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 0-28 (the output size of SHA-512/224 is 28 bytes). FIPS-198 compliant tokens may constrain the output length to be at least 4 or 14 (half the maximum length).

11840 Signatures (MACs) produced by this mechanism will be taken from the start of the full 28-byte HMAC
11841 output.

11842 *Table 155, 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

11843

11844 **6.25.4 SHA-512/224-HMAC**

11845 The SHA-512-HMAC mechanism, denoted **CKM_SHA512_224_HMAC**, is a special case of the general-
11846 length SHA-512/224-HMAC mechanism.

11847 It has no parameter, and always produces an output of length 28.

11848 **6.25.5 SHA-512/224 key derivation**

11849 The SHA-512/224 key derivation, denoted **CKM_SHA512_224_KEY_DERIVATION**, is the same as the
11850 SHA-512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/224 hash function
11851 and the relevant length is 28 bytes.

11852 **6.25.6 SHA-512/224 HMAC key generation**

11853 The SHA-512/224-HMAC key generation mechanism, denoted **CKM_SHA512_224_KEY_GEN**, is a key
11854 generation mechanism for NIST's SHA512/224-HMAC.

11855 It does not have a parameter.

11856 The mechanism generates SHA512/224-HMAC keys with a particular length in bytes, as specified in the
11857 **CKA_VALUE_LEN** attribute of the template for the key.

11858 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11859 key. Other attributes supported by the SHA512/224-HMAC key type (specifically, the flags indicating
11860 which functions the key supports) may be specified in the template for the key, or else are assigned
11861 default initial values.

11862 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11863 specify the supported range of **CKM_SHA512_224_HMAC** key sizes, in bytes.

11864 **6.26 SHA-512/256**

11865 *Table 156, SHA-512/256 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA512_256				✓				
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 157, 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.

Table 158, 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

6.26.4 SHA-512/256-HMAC

The SHA-512-HMAC mechanism, denoted **CKM_SHA512_256_HMAC**, is a special case of the general-length SHA-512/256-HMAC mechanism.

It has no parameter, and always produces an output of length 32.

6.26.5 SHA-512/256 key derivation

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.

6.26.6 SHA-512/256 HMAC key generation

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.

It does not have a parameter.

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.

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.

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.

6.27 SHA-512/t

Table 159, SHA-512 / t Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA512_T				✓				
CKM_SHA512_T_HMAC_GENERAL		✓						
CKM_SHA512_T_HMAC		✓						
CKM_SHA512_T_KEY_DERIVATION							✓	
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.

11933 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in
11934 bytes of the desired output should be in the range of $0-\lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$.

11935 Constraints on the length of input and output data are summarized in the following table. For single-part
11936 digesting, the data and the digest may begin at the same location in memory.

11937 *Table 160, SHA-512/256: Data Length*

Function	Input length	Digest length
C_Digest	any	$\lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$

11938 **6.27.3 General-length SHA-512/t-HMAC**

11939 The general-length SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC_GENERAL**, is the
11940 same as the general-length SHA-1-HMAC mechanism in section 6.20.3, except that it uses the HMAC
11941 construction based on the SHA-512/t hash function and length of the output should be in the range $0 -$
11942 $\lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$.

11943 **6.27.4 SHA-512/t-HMAC**

11944 The SHA-512/t-HMAC mechanism, denoted **CKM_SHA512_T_HMAC**, is a special case of the general-
11945 length SHA-512/t-HMAC mechanism.
11946 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the value of t in bits. The length in
11947 bytes of the desired output should be in the range of $0-\lceil t/8 \rceil$, where $0 < t < 512$, and $t \neq 384$.

11948 **6.27.5 SHA-512/t key derivation**

11949 The SHA-512/t key derivation, denoted **CKM_SHA512_T_KEY_DERIVATION**, is the same as the SHA-
11950 512 key derivation mechanism in section 6.24.5, except that it uses the SHA-512/t hash function and the
11951 relevant length is $\lceil t/8 \rceil$ bytes, where $0 < t < 512$, and $t \neq 384$.

11952 **6.27.6 SHA-512/t HMAC key generation**

11953 The SHA-512/t-HMAC key generation mechanism, denoted **CKM_SHA512_T_KEY_GEN**, is a key
11954 generation mechanism for NIST's SHA512/t-HMAC.
11955 It does not have a parameter.
11956 The mechanism generates SHA512/t-HMAC keys with a particular length in bytes, as specified in the
11957 **CKA_VALUE_LEN** attribute of the template for the key.
11958 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
11959 key. Other attributes supported by the SHA512/t-HMAC key type (specifically, the flags indicating which
11960 functions the key supports) may be specified in the template for the key, or else are assigned default
11961 initial values.
11962 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
11963 specify the supported range of **CKM_SHA512_T_HMAC** key sizes, in bytes.
11964

11965 **6.28 SHA3-224**

11966 *Table 161, SHA3-224 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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 162, 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.

11994 Table 163, 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

11995 6.28.4 SHA3-224-HMAC

11996 The SHA3-224-HMAC mechanism, denoted **CKM_SHA3_224_HMAC**, is a special case of the general-
11997 length SHA3-224-HMAC mechanism.

11998 It has no parameter, and always produces an output of length 28.

11999 6.28.5 SHA3-224 key derivation

12000 SHA-224 key derivation, denoted **CKM_SHA3_224_KEY_DERIVATION**, is the same as the SHA-1 key
12001 derivation mechanism in section 6.20.5 except that it uses the SHA3-224 hash function and the relevant
12002 length is 28 bytes.

12003 6.28.6 SHA3-224 HMAC key generation

12004 The SHA3-224-HMAC key generation mechanism, denoted **CKM_SHA3_224_KEY_GEN**, is a key
12005 generation mechanism for NIST's SHA3-224-HMAC.

12006 It does not have a parameter.

12007 The mechanism generates SHA3-224-HMAC keys with a particular length in bytes, as specified in the
12008 **CKA_VALUE_LEN** attribute of the template for the key.

12009 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12010 key. Other attributes supported by the SHA3-224-HMAC key type (specifically, the flags indicating which
12011 functions the key supports) may be specified in the template for the key, or else are assigned default
12012 initial values.

12013 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12014 specify the supported range of **CKM_SHA3_224_HMAC** key sizes, in bytes.

12015 6.29 SHA3-256

12016 Table 164, SHA3-256 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA3_256				✓				
CKM_SHA3_256_HMAC_GENERAL		✓						
CKM_SHA3_256_HMAC		✓						
CKM_SHA3_256_KEY_DERIVATION							✓	
CKM_SHA3_256_KEY_GEN					✓			

12017 6.29.1 Definitions

12018 Mechanisms:

12019 CKM_SHA3_256

- 12020CKM_SHA3_256_HMAC
- 12021CKM_SHA3_256_HMAC_GENERAL
- 12022CKM_SHA3_256_KEY_DERIVATION
- 12023CKM_SHA3_256_KEY_GEN
- 12024
- 12025CKK_SHA3_256_HMAC

12026

6.29.2 SHA3-256 digest

- 12027The SHA3-256 mechanism, denoted **CKM_SHA3_256**, is a mechanism for message digesting, following
- 12028the Secure Hash 3 Algorithm with a 256-bit message digest defined in [FIPS PUB 202].
- 12029It does not have a parameter.
- 12030Constraints on the length of input and output data are summarized in the following table. For single-part
- 12031digesting, the data and the digest may begin at the same location in memory.
- 12032Table 165, SHA3-256: Data Length

Function	Input length	Digest length
C_Digest	any	32

12033

6.29.3 General-length SHA3-256-HMAC

- 12034The general-length SHA3-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC_GENERAL**, is the
- 12035same as the general-length SHA-1-HMAC mechanism in section 6.20.4, except that it uses the HMAC
- 12036construction based on the SHA3-256 hash function and length of the output should be in the range 1-32.
- 12037The keys it uses are generic secret keys and **CKK_SHA3_256_HMAC**. FIPS-198 compliant tokens may
- 12038require the key length to be at least 16 bytes; that is, half the size of the SHA3-256 hash output.
- 12039It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
- 12040output. This length should be in the range 1-32 (the output size of SHA3-256 is 32 bytes). FIPS-198
- 12041compliant tokens may constrain the output length to be at least 4 or 16 (half the maximum length).
- 12042Signatures (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC
- 12043output.
- 12044Table 166, 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

12045

6.29.4 SHA3-256-HMAC

- 12046The SHA-256-HMAC mechanism, denoted **CKM_SHA3_256_HMAC**, is a special case of the general-
- 12047length SHA-256-HMAC mechanism.
- 12048It has no parameter, and always produces an output of length 32.

12049

6.29.5 SHA3-256 key derivation

- 12050SHA-256 key derivation, denoted **CKM_SHA3_256_KEY_DERIVATION**, is the same as the SHA-1 key
- 12051derivation mechanism in section 6.20.5, except that it uses the SHA3-256 hash function and the relevant
- 12052length is 32 bytes.

6.29.6 SHA3-256 HMAC key generation

The SHA3-256-HMAC key generation mechanism, denoted **CKM_SHA3_256_KEY_GEN**, is a key generation mechanism for NIST's SHA3-256-HMAC.

It does not have a parameter.

The mechanism generates SHA3-256-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 SHA3-256-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_SHA3_256_HMAC** key sizes, in bytes.

6.30 SHA3-384

Table 167, SHA3-384 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA3_384				✓				
CKM_SHA3_384_HMAC_GENERAL		✓						
CKM_SHA3_384_HMAC		✓						
CKM_SHA3_384_KEY_DERIVATION							✓	
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.

12082 Table 168, SHA3-384: Data Length

Function	Input length	Digest length
C_Digest	any	48

12083 **6.30.3 General-length SHA3-384-HMAC**

12084 The general-length SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC_GENERAL**, is the
12085 same as the general-length SHA-1-HMAC mechanism in section 6.20.4, except that it uses the HMAC
12086 construction based on the SHA-384 hash function and length of the output should be in the range 1-
12087 48. The keys it uses are generic secret keys and **CKK_SHA3_384_HMAC**. FIPS-198 compliant tokens
12088 may require the key length to be at least 24 bytes; that is, half the size of the SHA3-384 hash output.

12089
12090 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
12091 output. This length should be in the range 1-48 (the output size of SHA3-384 is 48 bytes). FIPS-198
12092 compliant tokens may constrain the output length to be at least 4 or 24 (half the maximum length).
12093 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 48-byte HMAC
12094 output.

12095 Table 169, 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

12096

12097 **6.30.4 SHA3-384-HMAC**

12098 The SHA3-384-HMAC mechanism, denoted **CKM_SHA3_384_HMAC**, is a special case of the general-
12099 length SHA3-384-HMAC mechanism.

12100 It has no parameter, and always produces an output of length 48.

12101 **6.30.5 SHA3-384 key derivation**

12102 SHA3-384 key derivation, denoted **CKM_SHA3_384_KEY_DERIVATION**, is the same as the SHA-1 key
12103 derivation mechanism in section 6.20.5, except that it uses the SHA-384 hash function and the relevant
12104 length is 48 bytes.

12105 **6.30.6 SHA3-384 HMAC key generation**

12106 The SHA3-384-HMAC key generation mechanism, denoted **CKM_SHA3_384_KEY_GEN**, is a key
12107 generation mechanism for NIST's SHA3-384-HMAC.

12108 It does not have a parameter.

12109 The mechanism generates SHA3-384-HMAC keys with a particular length in bytes, as specified in the
12110 **CKA_VALUE_LEN** attribute of the template for the key.

12111 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12112 key. Other attributes supported by the SHA3-384-HMAC key type (specifically, the flags indicating which
12113 functions the key supports) may be specified in the template for the key, or else are assigned default
12114 initial values.

12115 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12116 specify the supported range of **CKM_SHA3_384_HMAC** key sizes, in bytes.

12117 **6.31 SHA3-512**

12118 *Table 170, SHA-512 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SHA3_512				✓				
CKM_SHA3_512_HMAC_GENERAL		✓						
CKM_SHA3_512_HMAC		✓						
CKM_SHA3_512_KEY_DERIVATION							✓	
CKM_SHA3_512_KEY_GEN					✓			

12119 **6.31.1 Definitions**

- 12120 CKM_SHA3_512
- 12121 CKM_SHA3_512_HMAC
- 12122 CKM_SHA3_512_HMAC_GENERAL
- 12123 CKM_SHA3_512_KEY_DERIVATION
- 12124 CKM_SHA3_512_KEY_GEN
- 12125
- 12126 CKK_SHA3_512_HMAC

12127 **6.31.2 SHA3-512 digest**

- 12128 The SHA3-512 mechanism, denoted **CKM_SHA3_512**, is a mechanism for message digesting, following
- 12129 the Secure Hash 3 Algorithm with a 512-bit message digest defined in [FIPS PUB 202].
- 12130 It does not have a parameter.
- 12131 Constraints on the length of input and output data are summarized in the following table. For single-part
- 12132 digesting, the data and the digest may begin at the same location in memory.
- 12133 *Table 171, SHA3-512: Data Length*

Function	Input length	Digest length
C_Digest	any	64

12134 **6.31.3 General-length SHA3-512-HMAC**

- 12135 The general-length SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC_GENERAL**, is the
- 12136 same as the general-length SHA-1-HMAC mechanism in section 6.20.4, except that it uses the HMAC
- 12137 construction based on the SHA3-512 hash function and length of the output should be in the range 1-
- 12138 64. The keys it uses are generic secret keys and **CKK_SHA3_512_HMAC**. FIPS-198 compliant tokens
- 12139 may require the key length to be at least 32 bytes; that is, half the size of the SHA3-512 hash output.
- 12140
- 12141 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
- 12142 output. This length should be in the range 1-64 (the output size of SHA3-512 is 64 bytes). FIPS-198
- 12143 compliant tokens may constrain the output length to be at least 4 or 32 (half the maximum length).
- 12144 Signatures (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC
- 12145 output.

12146 Table 172, 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

12147

12148 6.31.4 SHA3-512-HMAC

12149 The SHA3-512-HMAC mechanism, denoted **CKM_SHA3_512_HMAC**, is a special case of the general-length SHA3-512-HMAC mechanism.

12151 It has no parameter, and always produces an output of length 64.

12152 6.31.5 SHA3-512 key derivation

12153 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.

12156 6.31.6 SHA3-512 HMAC key generation

12157 The SHA3-512-HMAC key generation mechanism, denoted **CKM_SHA3_512_KEY_GEN**, is a key generation mechanism for NIST's SHA3-512-HMAC.

12159 It does not have a parameter.

12160 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.

12162 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new key. Other attributes supported by the SHA3-512-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.

12166 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of **CKM_SHA3_512_HMAC** key sizes, in bytes.

12168 6.32 SHAKE

12169 Table 173, SHA-512 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_SHAKE_128_KEY_DERIVATION							✓	
CKM_SHAKE_256_KEY_DERIVATION							✓	

12170 6.32.1 Definitions

12171 CKM_SHAKE_128_KEY_DERIVATION

12172 CKM_SHAKE_256_KEY_DERIVATION

6.32.2 SHAKE Key Derivation

SHAKE-128 and SHAKE-256 key derivation, denoted **CKM_SHAKE_128_KEY_DERIVATION** and **CKM_SHAKE_256_KEY_DERIVATION**, implements the SHAKE expansion function defined in [FIPS PUB 202] on the input key.

- If no length or key type is provided in the template a **CKR_TEMPLATE_INCOMPLETE** error is generated.
- If no key type is provided in the template, but a length is, then the key produced by this mechanism shall 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 shall be of the type specified in the template. If it doesn't, an error shall 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 shall 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 shall be set properly.

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 shall 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 shall, 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.

6.33 BLAKE2B-160

Table 174, BLAKE2B-160 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_BLAKE2B_160				✓				
CKM_BLAKE2B_160_HMAC		✓						
CKM_BLAKE2B_160_HMAC_GENERAL		✓						
CKM_BLAKE2B_160_KEY_DERIVE							✓	
CKM_BLAKE2B_160_KEY_GEN					✓			

6.33.1 Definitions

Mechanisms:

- CKM_BLAKE2B_160
- CKM_BLAKE2B_160_HMAC
- CKM_BLAKE2B_160_HMAC_GENERAL

- 12207
- CKM_BLAKE2B_160_KEY_DERIVE
- 12208
- CKM_BLAKE2B_160_KEY_GEN
- 12209
- CKK_BLAKE2B_160_HMAC

12210

6.33.2 BLAKE2B-160 digest

- 12211
- The BLAKE2B-160 mechanism, denoted **CKM_BLAKE2B_160**, is a mechanism for message digesting,
- 12212
- following the Blake2b Algorithm with a 160-bit message digest without a key as defined in [\[RFC 7693\]](#).
- 12213
- It does not have a parameter.
- 12214
- Constraints on the length of input and output data are summarized in the following table. For single-part
- 12215
- digesting, the data and the digest may begin at the same location in memory.
- 12216
- Table 175, BLAKE2B-160: Data Length

Function	Input length	Digest length
C_Digest	any	20

12217

6.33.3 General-length BLAKE2B-160-HMAC

- 12218
- The general-length BLAKE2B-160-HMAC mechanism, denoted
- 12219
- CKM_BLAKE2B_160_HMAC_GENERAL**, is the keyed variant of BLAKE2b-160 and length of the output
- 12220
- should be in the range 1-20. The keys it uses are generic secret keys and **CKK_BLAKE2B_160_HMAC**.
- 12221
- It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
- 12222
- output. This length should be in the range 1-20 (the output size of BLAKE2B-160 is 20 bytes). Signatures
- 12223
- (MACs) produced by this mechanism shall be taken from the start of the full 20-byte HMAC output.
- 12224
- Table 176, General-length BLAKE2B-160-HMAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret or CKK_BLAKE2B_160_HMAC	Any	1-20, depending on parameters
C_Verify	generic secret or CKK_BLAKE2B_160_HMAC	Any	1-20, depending on parameters

12225

6.33.4 BLAKE2B-160-HMAC

- 12226
- The BLAKE2B-160-HMAC mechanism, denoted **CKM_BLAKE2B_160_HMAC**, is a special case of the
- 12227
- general-length BLAKE2B-160-HMAC mechanism.
- 12228
- It has no parameter, and always produces an output of length 20.

12229

6.33.5 BLAKE2B-160 key derivation

- 12230
- BLAKE2B-160 key derivation, denoted **CKM_BLAKE2B_160_KEY_DERIVE**, is the same as the SHA-1
- 12231
- key derivation mechanism in section 6.20.5 except that it uses the BLAKE2B-160 hash function and the
- 12232
- relevant length is 20 bytes.

12233

6.33.6 BLAKE2B-160 HMAC key generation

- 12234
- The BLAKE2B-160-HMAC key generation mechanism, denoted **CKM_BLAKE2B_160_KEY_GEN**, is a
- 12235
- key generation mechanism for BLAKE2B-160-HMAC.
- 12236
- It does not have a parameter.
- 12237
- The mechanism generates BLAKE2B-160-HMAC keys with a particular length in bytes, as specified in the
- 12238
- CKA_VALUE_LEN** attribute of the template for the key.
- 12239
- The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
- 12240
- key. Other attributes supported by the BLAKE2B-160-HMAC key type (specifically, the flags indicating

12241 which functions the key supports) may be specified in the template for the key, or else are assigned
12242 default initial values.
12243 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12244 specify the supported range of **CKM_BLAKE2B_160_HMAC** key sizes, in bytes.

12245 **6.34 BLAKE2B-256**

12246 *Table 177, BLAKE2B-256 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_BLAKE2B_256				✓				
CKM_BLAKE2B_256_HMAC_GENERAL		✓						
CKM_BLAKE2B_256_HMAC		✓						
CKM_BLAKE2B_256_KEY_DERIVE							✓	
CKM_BLAKE2B_256_KEY_GEN					✓			

12247 **6.34.1 Definitions**

- 12248 Mechanisms:
- 12249 CKM_BLAKE2B_256
 - 12250 CKM_BLAKE2B_256_HMAC
 - 12251 CKM_BLAKE2B_256_HMAC_GENERAL
 - 12252 CKM_BLAKE2B_256_KEY_DERIVE
 - 12253 CKM_BLAKE2B_256_KEY_GEN
 - 12254 CKK_BLAKE2B_256_HMAC

12255 **6.34.2 BLAKE2B-256 digest**

12256 The BLAKE2B-256 mechanism, denoted **CKM_BLAKE2B_256**, is a mechanism for message digesting,
12257 following the Blake2b Algorithm with a 256-bit message digest without a key as defined in [RFC 7693].
12258 It does not have a parameter.
12259 Constraints on the length of input and output data are summarized in the following table. For single-part
12260 digesting, the data and the digest may begin at the same location in memory.

12261 *Table 178, BLAKE2B-256: Data Length*

Function	Input length	Digest length
C_Digest	any	32

12262 **6.34.3 General-length BLAKE2B-256-HMAC**

12263 The general-length BLAKE2B-256-HMAC mechanism, denoted
12264 **CKM_BLAKE2B_256_HMAC_GENERAL**, is the keyed variant of Blake2b-256 and length of the output
12265 should be in the range 1-32. The keys it uses are generic secret keys and **CKK_BLAKE2B_256_HMAC**.
12266 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
12267 output. This length should be in the range 1-32 (the output size of BLAKE2B-256 is 32 bytes). Signatures
12268 (MACs) produced by this mechanism shall be taken from the start of the full 32-byte HMAC output.

12269 Table 179, 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

12270 6.34.4 BLAKE2B-256-HMAC

12271 The BLAKE2B-256-HMAC mechanism, denoted **CKM_BLAKE2B_256_HMAC**, is a special case of the
12272 general-length BLAKE2B-256-HMAC mechanism in section 6.34.3.

12273 It has no parameter, and always produces an output of length 32.

12274 6.34.5 BLAKE2B-256 key derivation

12275 BLAKE2B-256 key derivation, denoted **CKM_BLAKE2B_256_KEY_DERIVE**, is the same as the SHA-1
12276 key derivation mechanism in section 6.20.5, except that it uses the BLAKE2B-256 hash function and the
12277 relevant length is 32 bytes.

12278 6.34.6 BLAKE2B-256 HMAC key generation

12279 The BLAKE2B-256-HMAC key generation mechanism, denoted **CKM_BLAKE2B_256_KEY_GEN**, is a
12280 key generation mechanism for BLAKE2B-256-HMAC.

12281 It does not have a parameter.

12282 The mechanism generates BLAKE2B-256-HMAC keys with a particular length in bytes, as specified in the
12283 **CKA_VALUE_LEN** attribute of the template for the key.

12284 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12285 key. Other attributes supported by the BLAKE2B-256-HMAC key type (specifically, the flags indicating
12286 which functions the key supports) may be specified in the template for the key, or else are assigned
12287 default initial values.

12288 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12289 specify the supported range of **CKM_BLAKE2B_256_HMAC** key sizes, in bytes.

12290 6.35 BLAKE2B-384

12291 Table 180, BLAKE2B-384 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_BLAKE2B_384				✓				
CKM_BLAKE2B_384_HMAC_GENERAL		✓						
CKM_BLAKE2B_384_HMAC		✓						
CKM_BLAKE2B_384_KEY_DERIVE							✓	
CKM_BLAKE2B_384_KEY_GEN				✓				

12292 6.35.1 Definitions

12293 CKM_BLAKE2B_384

12294 CKM_BLAKE2B_384_HMAC
12295 CKM_BLAKE2B_384_HMAC_GENERAL
12296 CKM_BLAKE2B_384_KEY_DERIVE
12297 CKM_BLAKE2B_384_KEY_GEN
12298 CKK_BLAKE2B_384_HMAC

12299 **6.35.2 BLAKE2B-384 digest**

12300 The BLAKE2B-384 mechanism, denoted **CKM_BLAKE2B_384**, is a mechanism for message digesting,
12301 following the Blake2b Algorithm with a 384-bit message digest without a key as defined in [RFC 7693].
12302 It does not have a parameter.
12303 Constraints on the length of input and output data are summarized in the following table. For single-part
12304 digesting, the data and the digest may begin at the same location in memory.

12305 *Table 181, BLAKE2B-384: Data Length*

Function	Input length	Digest length
C_Digest	any	48

12306 **6.35.3 General-length BLAKE2B-384-HMAC**

12307 The general-length BLAKE2B-384-HMAC mechanism, denoted
12308 **CKM_BLAKE2B_384_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-384 hash function and
12309 length of the output should be in the range 1-48. The keys it uses are generic secret keys and
12310 **CKK_BLAKE2B_384_HMAC**.

12311
12312 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired output.
12313 This length should be in the range 1-48 (the output size of BLAKE2B-384 is 48 bytes). Signatures (MACs)
12314 produced by this mechanism shall be taken from the start of the full 48-byte HMAC output.

12315 *Table 182, 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

12316

12317 **6.35.4 BLAKE2B-384-HMAC**

12318 The BLAKE2B-384-HMAC mechanism, denoted **CKM_BLAKE2B_384_HMAC**, is a special case of the
12319 general-length BLAKE2B-384-HMAC mechanism.
12320 It has no parameter, and always produces an output of length 48.

12321 **6.35.5 BLAKE2B-384 key derivation**

12322 BLAKE2B-384 key derivation, denoted **CKM_BLAKE2B_384_KEY_DERIVE**, is the same as the SHA-1
12323 key derivation mechanism in section 6.20.5, except that it uses the BLAKE2B-384 hash function and the
12324 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 183, SHA-512 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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.

12352 Table 184, BLAKE2B-512: Data Length

Function	Input length	Digest length
C_Digest	any	64

12353 **6.36.3 General-length BLAKE2B-512-HMAC**

12354 The general-length BLAKE2B-512-HMAC mechanism, denoted
12355 **CKM_BLAKE2B_512_HMAC_GENERAL**, is the keyed variant of the BLAKE2B-512 hash function and
12356 length of the output should be in the range 1-64. The keys it uses are generic secret keys and
12357 **CKK_BLAKE2B_512_HMAC**.

12358
12359 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which holds the length in bytes of the desired
12360 output. This length should be in the range 1-64 (the output size of BLAKE2B-512 is 64 bytes). Signatures
12361 (MACs) produced by this mechanism shall be taken from the start of the full 64-byte HMAC output.

12362 Table 185, 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

12363

12364 **6.36.4 BLAKE2B-512-HMAC**

12365 The BLAKE2B-512-HMAC mechanism, denoted **CKM_BLAKE2B_512_HMAC**, is a special case of the
12366 general-length BLAKE2B-512-HMAC mechanism.
12367 It has no parameter, and always produces an output of length 64.

12368 **6.36.5 BLAKE2B-512 key derivation**

12369 BLAKE2B-512 key derivation, denoted **CKM_BLAKE2B_512_KEY_DERIVE**, is the same as the SHA-1
12370 key derivation mechanism in section 6.20.5, except that it uses the BLAKE2B-512 hash function and the
12371 relevant length is 64 bytes.

12372 **6.36.6 BLAKE2B-512 HMAC key generation**

12373 The BLAKE2B-512-HMAC key generation mechanism, denoted **CKM_BLAKE2B_512_KEY_GEN**, is a
12374 key generation mechanism for NIST's BLAKE2B-512-HMAC.

12375 It does not have a parameter.

12376 The mechanism generates BLAKE2B-512-HMAC keys with a particular length in bytes, as specified in the
12377 **CKA_VALUE_LEN** attribute of the template for the key.

12378 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12379 key. Other attributes supported by the BLAKE2B-512-HMAC key type (specifically, the flags indicating
12380 which functions the key supports) may be specified in the template for the key, or else are assigned
12381 default initial values.

12382 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12383 specify the supported range of **CKM_BLAKE2B_512_HMAC** key sizes, in bytes.

12384

6.37 PKCS #5 and PKCS #5-style password-based encryption (PBE)

The mechanisms in this section are for generating keys and IVs for performing password-based encryption. The method used to generate keys and IVs is specified in [PKCS #5].

Table 186, PKCS 5 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_PBE_SHA1_DES3_EDE_CBC					✓			
CKM_PBE_SHA1_DES2_EDE_CBC					✓			
CKM_PBA_SHA1_WITH_SHA1_HMAC					✓			
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;

12427 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE_PTR** is a pointer to a
12428 **CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE**.
12429

12430 ♦ **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;**
12431 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR**

12432 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE** is used to indicate the source of the salt value when
12433 deriving a key using PKCS #5 PBKDF2. It is defined as follows:

12434 typedef CK_ULONG CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE;

12435

12436 The following salt value sources are defined in [PKCS #5] v2.1. The following table lists the defined
12437 sources along with the corresponding data type for the *pSaltSourceData* field in the
12438 **CK_PKCS5_PBKD2_PARAMS2** structure defined below.

12439 *Table 188, 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.

12440 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE_PTR** is a pointer to a
12441 **CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE**.

12442 ♦ **CK_PKCS5_PBKD2_PARAMS2;**
12443 **CK_PKCS5_PBKD2_PARAMS2_PTR**

12444 **CK_PKCS5_PBKD2_PARAMS2** is a structure that provides the parameters to the
12445 **CKM_PKCS5_PBKD2** mechanism. The structure is defined as follows:

```
12446     typedef struct CK_PKCS5_PBKD2_PARAMS2 {  
12447         CK_PKCS5_PBKDF2_SALT_SOURCE_TYPE    saltSource;  
12448         CK_VOID_PTR                          pSaltSourceData;  
12449         CK_ULONG                            ulSaltSourceDataLen;  
12450         CK_ULONG                            iterations;  
12451         CK_PKCS5_PBKD2_PSEUDO_RANDOM_FUNCTION_TYPE prf;  
12452         CK_VOID_PTR                          pPrfData;  
12453         CK_ULONG                            ulPrfDataLen;  
12454         CK_UTF8CHAR_PTR                     pPassword;  
12455         CK_ULONG                            ulPasswordLen;  
12456     } CK_PKCS5_PBKD2_PARAMS2;
```

12457

12458 The fields of the structure have the following meanings:

12459	saltSource	source of the salt value
12460	pSaltSourceData	data used as the input for the salt source
12461	ulSaltSourceDataLen	length of the salt source input
12462	iterations	number of iterations to perform when generating each block of random data
12463		
12464	prf	pseudo-random function used to generate the key
12465	pPrfData	data used as the input for PRF in addition to the salt value

12466	ulPrfDataLen	length of the input data for the PRF
12467	pPassword	points to the password to be used in the PBE key generation
12468	ulPasswordLen	length in bytes of the password information

12469 **CK_PKCS5_PBKD2_PARAMS2_PTR** is a pointer to a **CK_PKCS5_PBKD2_PARAMS2**.

12470 6.37.4 PKCS #5 PBKD2 key generation

12471 PKCS #5 PBKDF2 key generation, denoted **CKM_PKCS5_PBKD2**, is a mechanism used for generating
 12472 a secret key from a password and a salt value. This functionality is defined in [PKCS #5] as PBKDF2.

12473 It has a parameter, a **CK_PKCS5_PBKD2_PARAMS2** structure. The parameter specifies the salt value
 12474 source, pseudo-random function, and iteration count used to generate the new key.

12475 Since this mechanism can be used to generate any type of secret key, new key templates must contain
 12476 the **CKA_KEY_TYPE** and **CKA_VALUE_LEN** attributes. If the key type has a fixed length the
 12477 **CKA_VALUE_LEN** attribute may be omitted.

12478 6.38 PKCS #12 password-based encryption/authentication 12479 mechanisms

12480 The mechanisms in this section are for generating keys and IVs for performing password-based
 12481 encryption or authentication. The method used to generate keys and IVs is based on a method that was
 12482 specified in [PKCS #12].

12483 We specify here a general method for producing various types of pseudo-random bits from a password,
 12484 p ; a string of salt bits, s ; and an iteration count, c . The “type” of pseudo-random bits to be produced is
 12485 identified by an identification byte, ID , the meaning of which will be discussed later.

12486 Let H be a hash function built around a compression function $f: \mathbf{Z}_2^u \times \mathbf{Z}_2^v \rightarrow \mathbf{Z}_2^u$ (that is, H has a chaining
 12487 variable and output of length u bits, and the message input to the compression function of H is v bits). For
 12488 MD2 and MD5, $u=128$ and $v=512$; for SHA-1, $u=160$ and $v=512$.

12489 We assume here that u and v are both multiples of 8, as are the lengths in bits of the password and salt
 12490 strings and the number n of pseudo-random bits required. In addition, u and v are of course nonzero.

- 12491 1. Construct a string, D (the “diversifier”), by concatenating $v/8$ copies of ID .
- 12492 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
 12493 salt may be truncated to create S). Note that if the salt is the empty string, then so is S .
- 12494 3. Concatenate copies of the password together to create a string P of length $v \lceil p/v \rceil$ bits (the final copy
 12495 of the password may be truncated to create P). Note that if the password is the empty string, then so
 12496 is P .
- 12497 4. Set $I = S || P$ to be the concatenation of S and P .
- 12498 5. Set $j = \lceil n/u \rceil$.
- 12499 6. For $i = 1, 2, \dots, j$, do the following:
 - 12500 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
 12501 that hash; etc.; continue in this fashion until a total of c hashes have been computed, each on
 12502 the result of the previous hash.
 - 12503 b. Concatenate copies of A_i to create a string B of length v bits (the final copy of A_i may be
 12504 truncated to create B).
 - 12505 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
 12506 setting $I_j = (I_j + B + 1) \bmod 2^v$ for each j . To perform this addition, treat each v -bit block as a
 12507 binary number represented most-significant bit first.
- 12508 7. Concatenate A_1, A_2, \dots, A_j together to form a pseudo-random bit string, A .
- 12509 8. Use the first n bits of A as the output of this entire process.

12510 When the password-based encryption mechanisms presented in this section are used to generate a key
12511 and IV (if needed) from a password, salt, and an iteration count, the above algorithm is used. To generate
12512 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
12513 2.

12514 When the password based authentication mechanism presented in this section is used to generate a key
12515 from a password, salt, and an iteration count, the above algorithm is used. The identifier byte *ID* is set to
12516 the value 3.

12517 6.38.1 SHA-1-PBE for 3-key triple-DES-CBC

12518 SHA-1-PBE for 3-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES3_EDE_CBC**, is a mechanism
12519 used for generating a 3-key triple-DES secret key and IV from a password and a salt value by using the
12520 SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described
12521 above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 3-
12522 key triple-DES key with proper parity bits is obtained.

12523 It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the
12524 key generation process and the location of the application-supplied buffer which will receive the 8-byte IV
12525 generated by the mechanism.

12526 The key and IV produced by this mechanism will typically be used for performing password-based
12527 encryption.

12528 6.38.2 SHA-1-PBE for 2-key triple-DES-CBC

12529 SHA-1-PBE for 2-key triple-DES-CBC, denoted **CKM_PBE_SHA1_DES2_EDE_CBC**, is a mechanism
12530 used for generating a 2-key triple-DES secret key and IV from a password and a salt value by using the
12531 SHA-1 digest algorithm and an iteration count. The method used to generate the key and IV is described
12532 above. Each byte of the key produced will have its low-order bit adjusted, if necessary, so that a valid 2-
12533 key triple-DES key with proper parity bits is obtained.

12534 It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the
12535 key generation process and the location of the application-supplied buffer which will receive the 8-byte IV
12536 generated by the mechanism.

12537 The key and IV produced by this mechanism will typically be used for performing password-based
12538 encryption.

12539 6.38.3 SHA-1-PBA for SHA-1-HMAC

12540 SHA-1-PBA for SHA-1-HMAC, denoted **CKM_PBA_SHA1_WITH_SHA1_HMAC**, is a mechanism used
12541 for generating a 160-bit generic secret key from a password and a salt value by using the SHA-1 digest
12542 algorithm and an iteration count. The method used to generate the key is described above.

12543 It has a parameter, a **CK_PBE_PARAMS** structure. The parameter specifies the input information for the
12544 key generation process. The parameter also has a field to hold the location of an application-supplied
12545 buffer which will receive an IV; for this mechanism, the contents of this field are ignored, since
12546 authentication with SHA-1-HMAC does not require an IV.

12547 The key generated by this mechanism will typically be used for computing a SHA-1 HMAC to perform
12548 password-based authentication (not *password-based encryption*). At the time of this writing, this is
12549 primarily done to ensure the integrity of a PKCS #12 PDU.

12550 6.39 SSL

12551 *Table 189, SSL Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SSL3_PRE_MASTER_KEY_GEN					✓			
CKM_TLS_PRE_MASTER_KEY_GEN					✓			
CKM_SSL3_MASTER_KEY_DERIVE							✓	
CKM_SSL3_MASTER_KEY_DERIVE_DH							✓	
CKM_SSL3_KEY_AND_MAC_DERIVE							✓	
CKM_SSL3_MD5_MAC		✓						
CKM_SSL3_SHA1_MAC		✓						

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

12578 ♦ **CK_SSL3_MASTER_KEY_DERIVE_PARAMS;**
12579 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR**

12580 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS** is a structure that provides the parameters to the
12581 **CKM_SSL3_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
12582     typedef struct CK_SSL3_MASTER_KEY_DERIVE_PARAMS {  
12583         CK_SSL3_RANDOM_DATA    RandomInfo;  
12584         CK_VERSION_PTR          pVersion;  
12585     } CK_SSL3_MASTER_KEY_DERIVE_PARAMS;
```

12586
12587 The fields of the structure have the following meanings:

12588	RandomInfo	client's and server's random data information.
12589	pVersion	pointer to a CK_VERSION structure which receives the SSL
12590		protocol version information

12591 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
12592 **CK_SSL3_MASTER_KEY_DERIVE_PARAMS**.

12593 ♦ **CK_SSL3_KEY_MAT_OUT; CK_SSL3_KEY_MAT_OUT_PTR**

12594 **CK_SSL3_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization vectors
12595 after performing a **C_DeriveKey** function with the **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It
12596 is defined as follows:

```
12597     typedef struct CK_SSL3_KEY_MAT_OUT {  
12598         CK_OBJECT_HANDLE    hClientMacSecret;  
12599         CK_OBJECT_HANDLE    hServerMacSecret;  
12600         CK_OBJECT_HANDLE    hClientKey;  
12601         CK_OBJECT_HANDLE    hServerKey;  
12602         CK_BYTE_PTR          pIVClient;  
12603         CK_BYTE_PTR          pIVServer;  
12604     } CK_SSL3_KEY_MAT_OUT;
```

12605
12606 The fields of the structure have the following meanings:

12607	hClientMacSecret	key handle for the resulting Client MAC Secret key
12608	hServerMacSecret	key handle for the resulting Server MAC Secret key
12609	hClientKey	key handle for the resulting Client Secret key
12610	hServerKey	key handle for the resulting Server Secret key
12611	pIVClient	pointer to a location which receives the initialization vector (IV)
12612		created for the client (if any)
12613	pIVServer	pointer to a location which receives the initialization vector (IV)
12614		created for the server (if any)

12615 **CK_SSL3_KEY_MAT_OUT_PTR** is a pointer to a **CK_SSL3_KEY_MAT_OUT**.

12616 ♦ **CK_SSL3_KEY_MAT_PARAMS;**
12617 **CK_SSL3_KEY_MAT_PARAMS_PTR**

12618 **CK_SSL3_KEY_MAT_PARAMS** is a structure that provides the parameters to the
12619 **CKM_SSL3_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```

12620     typedef struct CK_SSL3_KEY_MAT_PARAMS {
12621         CK_ULONG          ulMacSizeInBits;
12622         CK_ULONG          ulKeySizeInBits;
12623         CK_ULONG          ulIVSizeInBits;
12624         CK_BBOOL          bIsExport;
12625         CK_SSL3_RANDOM_DATA RandomInfo;
12626         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
12627     } CK_SSL3_KEY_MAT_PARAMS;

```

12628

12629 The fields of the structure have the following meanings:

12630	ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the
12631		protocol handshake phase
12632	ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the
12633		protocol handshake phase
12634	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol
12635		handshake phase. If no IV is required, the length should be set to 0
12636	bIsExport	a Boolean value which indicates whether the keys have to be
12637		derived for an export version of the protocol
12638	RandomInfo	client's and server's random data information.
12639	pReturnedKeyMaterial	points to a CK_SSL3_KEY_MAT_OUT structures which receives
12640		the handles for the keys generated and the IVs

12641 **CK_SSL3_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_SSL3_KEY_MAT_PARAMS**.

12642 6.39.3 Pre-master key generation

12643 Pre-master key generation in SSL 3.0, denoted **CKM_SSL3_PRE_MASTER_KEY_GEN**, is a mechanism
12644 which generates a 48-byte generic secret key. It is used to produce the "pre_master" key used in SSL
12645 version 3.0 for RSA-like cipher suites.

12646 It has one parameter, a **CK_VERSION** structure, which provides the client's SSL version number.

12647 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
12648 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
12649 be specified in the template, or else are assigned default values.

12650 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
12651 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
12652 attribute has value 48. However, since these facts are all implicit in the mechanism, there is no need to
12653 specify any of them.

12654 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the **CK_MECHANISM_INFO** structure
12655 both indicate 48 bytes.

12656 **CKM_TLS_PRE_MASTER_KEY_GEN** has identical functionality as
12657 **CKM_SSL3_PRE_MASTER_KEY_GEN**. It exists only for historical reasons, please use
12658 **CKM_SSL3_PRE_MASTER_KEY_GEN** instead.

12659 6.39.4 Master key derivation

12660 Master key derivation in SSL 3.0, denoted **CKM_SSL3_MASTER_KEY_DERIVE**, is a mechanism used
12661 to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce
12662 the "master_secret" key used in the SSL protocol from the "pre_master" key. This mechanism returns the
12663 value of the client version, which is built into the "pre_master" key as well as a handle to the derived
12664 "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.

12765 If a call to **C_DeriveKey** with this mechanism fails, then *none* of the four keys will be created on the
12766 token.

12767 **6.39.7 MD5 MACing in SSL 3.0**

12768 MD5 MACing in SSL3.0, denoted **CKM_SSL3_MD5_MAC**, is a mechanism for single- and multiple-part
12769 signatures (data authentication) and verification using MD5, based on the SSL 3.0 protocol. This
12770 technique is very similar to the HMAC technique.

12771 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
12772 signatures produced by this mechanism.

12773 Constraints on key types and the length of input and output data are summarized in the following table:

12774 *Table 190, 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

12775 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12776 specify the supported range of generic secret key sizes, in bits.

12777 **6.39.8 SHA-1 MACing in SSL 3.0**

12778 SHA-1 MACing in SSL3.0, denoted **CKM_SSL3_SHA1_MAC**, is a mechanism for single- and multiple-
12779 part signatures (data authentication) and verification using SHA-1, based on the SSL 3.0 protocol. This
12780 technique is very similar to the HMAC technique.

12781 It has a parameter, a **CK_MAC_GENERAL_PARAMS**, which specifies the length in bytes of the
12782 signatures produced by this mechanism.

12783 Constraints on key types and the length of input and output data are summarized in the following table:

12784 *Table 191, 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

12785 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
12786 specify the supported range of generic secret key sizes, in bits.

12787 **6.40 TLS 1.2 Mechanisms**

12788 Details for TLS 1.2 and its key derivation and MAC mechanisms can be found in [TLS12]. TLS 1.2
12789 mechanisms differ from TLS 1.0 and 1.1 mechanisms in that the base hash used in the underlying TLS
12790 PRF (pseudo-random function) can be negotiated. Therefore each mechanism parameter for the TLS 1.2
12791 mechanisms contains a new value in the parameters structure to specify the hash function.

12792 This section also specifies **CKM_TLS12_MAC** which should be used in place of **CKM_TLS_PRF** to
12793 calculate the *verify_data* in the TLS "finished" message.

12794 This section also specifies **CKM_TLS_KDF** (and **CKM_TLS12_KDF**) that can be used in place of
12795 **CKM_TLS_PRF** to implement key material exporters.

12796

12797 *Table 192, TLS 1.2 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_TLS12_MASTER_KEY_DERIVE							✓	
CKM_TLS12_MASTER_KEY_DERIVE_DH							✓	
CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE							✓	
CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE_DH							✓	
CKM_TLS12_KEY_AND_MAC_DERIVE							✓	
CKM_TLS12_KEY_SAFE_DERIVE							✓	
CKM_TLS_MAC		✓						
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_EXTENDED_MASTER_KEY_DERIVE
- CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE_DH
- CKM_TLS12_KEY_AND_MAC_DERIVE
- CKM_TLS12_KEY_SAFE_DERIVE
- CKM_TLS_MAC
- 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:

12822	RandomInfo	client's and server's random data information as specified in section 6.39.2.
12823		
12824	pVersion	pointer to a CK_VERSION structure which receives the SSL protocol version information.
12825		
12826	prfHashMechanism	base hash used in the underlying TLS1.2 PRF operation used to derive the master key.
12827		

12828

12829 **CK_TLS12_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
 12830 **CK_TLS12_MASTER_KEY_DERIVE_PARAMS**.

12831 ♦ **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS;**
 12832 **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS_PTR**

12833 **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS** is a structure that provides the parameters
 12834 to the **CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```

12835     typedef struct CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS {
12836         CK_MECHANISM_TYPE prfHashMechanism;
12837         CK_BYTE_PTR pSessionHash;
12838         CK_ULONG ulSessionHashLen;
12839         CK_VERSION_PTR pVersion;
12840     } CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS;
  
```

12841

12842 The fields of the structure have the following meanings:

12843	prfHashMechanism	base hash used in the underlying TLS1.2 PRF operation used to derive the master key.
12844		
12845	pSessionHash	pointer to the session hash data defined in [RFC 7627].
12846	ulSessionHashLen	length of the data pointed to by pSessionHash.
12847	pVersion	pointer to a CK_VERSION structure which receives the SSL protocol version information
12848		

12849

12850 **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
 12851 **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS**.

12852 ♦ **CK_TLS12_KEY_MAT_PARAMS;**
 12853 **CK_TLS12_KEY_MAT_PARAMS_PTR**

12854 **CK_TLS12_KEY_MAT_PARAMS** is a structure that provides the parameters to the
 12855 **CKM_TLS12_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```

12856     typedef struct CK_TLS12_KEY_MAT_PARAMS {
12857         CK_ULONG ulMacSizeInBits;
12858         CK_ULONG ulKeySizeInBits;
12859         CK_ULONG ulIVSizeInBits;
12860         CK_BBOOL bIsExport;
12861         CK_SSL3_RANDOM_DATA RandomInfo;
12862         CK_SSL3_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
12863         CK_MECHANISM_TYPE prfHashMechanism;
12864     } CK_TLS12_KEY_MAT_PARAMS;
  
```

12865		
12866	The fields of the structure have the following meanings:	
12867	ulMacSizeInBits	the length (in bits) of the MACing keys agreed upon during the
12868		protocol handshake phase. If no MAC key is required, the length
12869		should be set to 0.
12870	ulKeySizeInBits	the length (in bits) of the secret keys agreed upon during the
12871		protocol handshake phase
12872	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the protocol
12873		handshake phase. If no IV is required, the length should be set to 0
12874	blsExport	must be set to CK_FALSE because export cipher suites must not be
12875		used in TLS 1.1 and later.
12876	RandomInfo	client's and server's random data information as specified in section
12877		6.39.2.
12878	pReturnedKeyMaterial	points to a CK_SSL3_KEY_MAT_OUT structure as specified in
12879		section 6.39.2 which receives the handles for the keys generated
12880		and the IVs
12881	prfHashMechanism	base hash used in the underlying TLS1.2 PRF operation used to
12882		derive the master key.

12883 **CK_TLS12_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_TLS12_KEY_MAT_PARAMS**.

12884 ♦ **CK_TLS_KDF_PARAMS; CK_TLS_KDF_PARAMS_PTR**

12885 **CK_TLS_KDF_PARAMS** is a structure that provides the parameters to the **CKM_TLS_KDF** (and
12886 **CKM_TLS12_KDF**) mechanism. It is defined as follows:

```
12887     typedef struct CK_TLS_KDF_PARAMS {
12888         CK_MECHANISM_TYPE prfMechanism;
12889         CK_BYTE_PTR pLabel;
12890         CK_ULONG ulLabelLength;
12891         CK_SSL3_RANDOM_DATA RandomInfo;
12892         CK_BYTE_PTR pContextData;
12893         CK_ULONG ulContextDataLength;
12894     } CK_TLS_KDF_PARAMS;
```

12895
12896 The fields of the structure have the following meanings:

12897	prfMechanism	the hash mechanism used in the TLS1.2 PRF construct or
12898		CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
12899	pLabel	a pointer to the label for this key derivation
12900	ulLabelLength	length of the label in bytes
12901	RandomInfo	the random data for the key derivation as specified in section 6.39.2
12902	pContextData	a pointer to the context data for this key derivation. NULL_PTR if not
12903		present
12904	ulContextDataLength	length of the context data in bytes. 0 if not present.

12905 **CK_TLS_KDF_PARAMS_PTR** is a pointer to a **CK_TLS_KDF_PARAMS**.

12906

♦ CK_TLS_MAC_PARAMS; CK_TLS_MAC_PARAMS_PTR

12907 **CK_TLS_MAC_PARAMS** is a structure that provides the parameters to the **CKM_TLS_MAC** (and
12908 **CKM_TLS12_MAC**) mechanism. It is defined as follows:

```
12909     typedef struct CK_TLS_MAC_PARAMS {  
12910         CK_MECHANISM_TYPE prfHashMechanism;  
12911         CK_ULONG ulMacLength;  
12912         CK_ULONG ulServerOrClient;  
12913     } CK_TLS_MAC_PARAMS;  
12914
```

12915 The fields of the structure have the following meanings:

12916	prfHashMechanism	the hash mechanism used in the TLS12 PRF construct or
12917		CKM_TLS_PRF to use with the TLS1.0 and 1.1 PRF construct.
12918	ulMacLength	the length of the MAC tag required or offered. Always 12 octets in
12919		TLS 1.0 and 1.1. Generally 12 octets, but may be negotiated to a
12920		longer value in TLS1.2.
12921	ulServerOrClient	1 to use the label "server finished", 2 to use the label "client
12922		finished". All other values are invalid.

12923 **CK_TLS_MAC_PARAMS_PTR** is a pointer to a **CK_TLS_MAC_PARAMS**.

12924

12925

♦ CK_TLS_PRF_PARAMS; CK_TLS_PRF_PARAMS_PTR

12926 **CK_TLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_TLS_PRF**
12927 mechanism. It is defined as follows:

```
12928     typedef struct CK_TLS_PRF_PARAMS {  
12929         CK_BYTE_PTR pSeed;  
12930         CK_ULONG ulSeedLen;  
12931         CK_BYTE_PTR pLabel;  
12932         CK_ULONG ulLabelLen;  
12933         CK_BYTE_PTR pOutput;  
12934         CK_ULONG_PTR pulOutputLen;  
12935     } CK_TLS_PRF_PARAMS;  
12936
```

12937 The fields of the structure have the following meanings:

12938	pSeed	pointer to the input seed
12939	ulSeedLen	length in bytes of the input seed
12940	pLabel	pointer to the identifying label
12941	ulLabelLen	length in bytes of the identifying label
12942	pOutput	pointer receiving the output of the operation
12943	pulOutputLen	pointer to the length in bytes that the output to be created shall
12944		have, has to hold the desired length as input and will receive the
12945		calculated length as output

12946 **CK_TLS_PRF_PARAMS_PTR** is a pointer to a **CK_TLS_PRF_PARAMS**.

6.40.3 TLS MAC

The TLS MAC mechanism is used to generate integrity tags for the TLS "finished" message. It replaces the use of the **CKM_TLS_PRF** function for TLS1.0 and 1.1 and that mechanism is deprecated.

CKM_TLS_MAC takes a parameter of **CK_TLS_MAC_PARAMS**. To use this mechanism with TLS1.0 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note: Although **CKM_TLS_PRF** is deprecated as a mechanism for **C_DeriveKey**, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 pseudo-random function.

In TLS1.0 and 1.1 the "finished" message *verify_data* (i.e. the output signature from the MAC mechanism) is always 12 bytes. In TLS1.2 the "finished" message *verify_data* is a minimum of 12 bytes, defaults to 12 bytes, but may be negotiated to longer length.

Note: **CKM_TLS12_MAC** is provided only for historical reasons and should be considered deprecated. This mechanism shares the same behavior with **CKM_TLS_MAC**.

Table 193, General-length TLS MAC: Key And Data Length

Function	Key type	Data length	Signature length
C_Sign	generic secret	any	≥ 12 bytes
C_Verify	generic secret	any	≥ 12 bytes

6.40.4 Master key derivation

Master key derivation in TLS 1.2, denoted **CKM_TLS12_MASTER_KEY_DERIVE**, is a mechanism used to derive one 48-byte generic secret key from another 48-byte generic secret key. It is used to produce the "master_secret" key used in the TLS protocol from the "pre_master" key. This mechanism returns the value of the client version, which is built into the "pre_master" key as well as a handle to the derived "master_secret" key. **CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE** is the same as **CKM_TLS12_MASTER_KEY_DERIVE** except it uses [RFC 7627] as the method to generate a new 48 byte generic secret key.

CKM_TLS12_MASTER_KEY_DERIVE has a parameter, a **CK_TLS12_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token, the underlying prf used in the key derivation, as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in section 6.40.2.

CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE has a parameter, a **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of session hash, the underlying prf used in the key derivation, as well as the returning of the protocol version number which is part of the pre-master key. This structure is defined in section 6.40.2.

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 mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS_KDF**, **CKM_TLS_MAC**. Additionally implementations that support them can add the deprecated **CKM_TLS12_KDF** and **CKM_TLS12_MAC** mechanisms.

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.

6.40.5 Master key derivation for Diffie-Hellman

Master key derivation for Diffie-Hellman in TLS 1.2, denoted **CKM_TLS12_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 TLS protocol from the "pre_master" key. **CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE_DH** is the same as **CKM_TLS12_MASTER_KEY_DERIVE_DH** except it uses [RFC 7627] as the method to generate a new 48 byte generic secret key.

CKM_TLS12_MASTER_KEY_DERIVE_DH has a parameter, a **CK_TLS12_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of random data to the token and the underlying prf used to generate the key. This structure is defined in section 6.40.2. 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.

CKM_TLS12_EXTENDED_MASTER_KEY_DERIVE_DH has a parameter, a **CK_TLS12_EXTENDED_MASTER_KEY_DERIVE_PARAMS** structure, which allows for the passing of session hash to the token and the underlying prf used to generate the key. This structure is defined in section 6.40.2. 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 mechanism also contributes the **CKA_ALLOWED_MECHANISMS** attribute consisting only of **CKM_TLS12_KEY_AND_MAC_DERIVE**, **CKM_TLS12_KEY_SAFE_DERIVE**, **CKM_TLS_KDF**, **CKM_TLS_MAC**. Additionally implementations that support them can add the deprecated **CKM_TLS12_KDF** and **CKM_TLS12_MAC** mechanisms.

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.

6.40.6 Key and MAC derivation

Key, MAC and IV derivation in TLS 1.2, denoted **CKM_TLS12_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 6.39.2.

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_TLS12_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 and treated as 0. All of the other conditions and behavior described for **CKM_TLS12_KEY_AND_MAC_DERIVE**, with the exception of the black box warning, apply to this mechanism.

CKM_TLS12_KEY_SAFE_DERIVE is provided as a separate mechanism to allow a client to control the export of IV material (and possible leaking of key material) through the use of the **CKA_ALLOWED_MECHANISMS** key attribute.

6.40.8 Generic Key Derivation using the TLS PRF

CKM_TLS_KDF is the mechanism defined in [RFC 5705]. It uses the TLS key material and TLS PRF function to produce additional key material for protocols that want to leverage the TLS key negotiation mechanism. **CKM_TLS_KDF** has a parameter of **CK_TLS_KDF_PARAMS**. If the protocol using this mechanism does not use context information, the *pContextData* field shall be set to **NULL_PTR** and the *ulContextDataLength* field shall be set to 0.

To use this mechanism with TLS1.0 and TLS1.1, use **CKM_TLS_PRF** as the value for *prfMechanism* in place of a hash mechanism. Note: Although **CKM_TLS_PRF** is deprecated as a mechanism for **C_DeriveKey**, the manifest value is retained for use with this mechanism to indicate the use of the TLS1.0/1.1 Pseudo-random function.

This mechanism can be used to derive multiple keys (e.g. similar to **CKM_TLS12_KEY_AND_MAC_DERIVE**) by first deriving the key stream as a **CKK_GENERIC_SECRET** of the necessary length and doing subsequent derives against that derived key using the **CKM_EXTRACT_KEY_FROM_KEY** mechanism to split the key stream into the actual operational keys.

The mechanism should not be used with the labels defined for use with TLS, but the token does not enforce this behavior.

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

- If the original 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 the original key.
- Similarly, if the original 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 the original key.
- The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to **CK_TRUE** if and only if the original 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 original key has its **CKA_NEVER_EXTRACTABLE** attribute set to **CK_TRUE**.

6.40.9 Deprecated TLS 1.2 mechanisms

CKM_TLS12_KDF and **CKM_TLS12_MAC** are considered aliases of the corresponding **CKM_TLS_KDF** and **CKM_TLS_MAC** mechanisms and behave identically. These mechanisms are available with a different identifier for historical reasons and are considered deprecated.

6.41 WTLS

Details can be found in [WTLS].

When comparing the existing TLS mechanisms with these extensions to support WTLS one could argue that there would be no need to have distinct handling of the client and server side of the handshake. However, since in WTLS the server and client use different sequence numbers, there could be instances (e.g. when WTLS is used to protect asynchronous protocols) where sequence numbers on the client and server side differ, and hence this motivates the introduced split.

Table 194, WTLS Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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							✓	

6.41.1 Definitions

Mechanisms:

CKM_WTLS_PRE_MASTER_KEY_GEN
CKM_WTLS_MASTER_KEY_DERIVE
CKM_WTLS_MASTER_KEY_DERIVE_DH_ECC
CKM_WTLS_PRF
CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE
CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE

6.41.2 WTLS mechanism parameters

◆ CK_WTLS_RANDOM_DATA; CK_WTLS_RANDOM_DATA_PTR

CK_WTLS_RANDOM_DATA is a structure, which provides information about the random data of a client and a server in a WTLS context. This structure is used by the **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```
typedef struct CK_WTLS_RANDOM_DATA {
```

```

13162     CK_BYTE_PTR pClientRandom;
13163     CK_ULONG     ulClientRandomLen;
13164     CK_BYTE_PTR pServerRandom;
13165     CK_ULONG     ulServerRandomLen;
13166 } CK_WTLS_RANDOM_DATA;
13167

```

13168 The fields of the structure have the following meanings:

```

13169         pClientRandom    pointer to the client's random data
13170         pClientRandomLen  length in bytes of the client's random data
13171         pServerRandom     pointer to the server's random data
13172         ulServerRandomLen length in bytes of the server's random data

```

13173 **CK_WTLS_RANDOM_DATA_PTR** is a pointer to a **CK_WTLS_RANDOM_DATA**.

13174 ♦ **CK_WTLS_MASTER_KEY_DERIVE_PARAMS;**
13175 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR**

13176 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS** is a structure, which provides the parameters to the
13177 **CKM_WTLS_MASTER_KEY_DERIVE** mechanism. It is defined as follows:

```

13178     typedef struct CK_WTLS_MASTER_KEY_DERIVE_PARAMS {
13179         CK_MECHANISM_TYPE    DigestMechanism;
13180         CK_WTLS_RANDOM_DATA  RandomInfo;
13181         CK_BYTE_PTR          pVersion;
13182     } CK_WTLS_MASTER_KEY_DERIVE_PARAMS;

```

13183

13184 The fields of the structure have the following meanings:

```

13185         DigestMechanism    the mechanism type of the digest mechanism to be used (possible
13186                             types can be found in [WTLS])
13187         RandomInfo         Client's and server's random data information
13188         pVersion            pointer to a CK_BYTE which receives the WTLS protocol version
13189                             information

```

13190 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS_PTR** is a pointer to a
13191 **CK_WTLS_MASTER_KEY_DERIVE_PARAMS**.

13192 ♦ **CK_WTLS_PRF_PARAMS; CK_WTLS_PRF_PARAMS_PTR**

13193 **CK_WTLS_PRF_PARAMS** is a structure, which provides the parameters to the **CKM_WTLS_PRF**
13194 mechanism. It is defined as follows:

```

13195     typedef struct CK_WTLS_PRF_PARAMS {
13196         CK_MECHANISM_TYPE    DigestMechanism;
13197         CK_BYTE_PTR          pSeed;
13198         CK_ULONG             ulSeedLen;
13199         CK_BYTE_PTR          pLabel;
13200         CK_ULONG             ulLabelLen;
13201         CK_BYTE_PTR          pOutput;
13202         CK_ULONG_PTR         pulOutputLen;
13203     } CK_WTLS_PRF_PARAMS;

```

13204		
13205	The fields of the structure have the following meanings:	
13206	Digest Mechanism	the mechanism type of the digest mechanism to be used (possible
13207		types can be found in [WTLS])
13208	pSeed	pointer to the input seed
13209	ulSeedLen	length in bytes of the input seed
13210	pLabel	pointer to the identifying label
13211	ulLabelLen	length in bytes of the identifying label
13212	pOutput	pointer receiving the output of the operation
13213	pulOutputLen	pointer to the length in bytes that the output to be created shall
13214		have, has to hold the desired length as input and will receive the
13215		calculated length as output

13216 **CK_WTLS_PRF_PARAMS_PTR** is a pointer to a **CK_WTLS_PRF_PARAMS**.

13217 ♦ **CK_WTLS_KEY_MAT_OUT; CK_WTLS_KEY_MAT_OUT_PTR**

13218 **CK_WTLS_KEY_MAT_OUT** is a structure that contains the resulting key handles and initialization
 13219 vectors after performing a **C_DeriveKey** function with the
 13220 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** or with the
 13221 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanism. It is defined as follows:

```

13222     typedef struct CK_WTLS_KEY_MAT_OUT {
13223         CK_OBJECT_HANDLE hMacSecret;
13224         CK_OBJECT_HANDLE hKey;
13225         CK_BYTE_PTR      pIV;
13226     } CK_WTLS_KEY_MAT_OUT;
  
```

13227
 13228 The fields of the structure have the following meanings:

13229	hMacSecret	Key handle for the resulting MAC secret key
13230	hKey	Key handle for the resulting secret key
13231	pIV	Pointer to a location which receives the initialization vector (IV)
13232		created (if any)

13233 **CK_WTLS_KEY_MAT_OUT_PTR** is a pointer to a **CK_WTLS_KEY_MAT_OUT**.

13234 ♦ **CK_WTLS_KEY_MAT_PARAMS; 13235 CK_WTLS_KEY_MAT_PARAMS_PTR**

13236 **CK_WTLS_KEY_MAT_PARAMS** is a structure that provides the parameters to the
 13237 **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE** and the
 13238 **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE** mechanisms. It is defined as follows:

```

13239     typedef struct CK_WTLS_KEY_MAT_PARAMS {
13240         CK_MECHANISM_TYPE    DigestMechanism;
13241         CK_ULONG              ulMacSizeInBits;
13242         CK_ULONG              ulKeySizeInBits;
13243         CK_ULONG              ulIVSizeInBits;
13244         CK_ULONG              ulSequenceNumber;
13245         CK_BBOOL              bIsExport;
  
```

```

13246         CK_WTLS_RANDOM_DATA        RandomInfo;
13247         CK_WTLS_KEY_MAT_OUT_PTR pReturnedKeyMaterial;
13248     } CK_WTLS_KEY_MAT_PARAMS;

```

13249

13250 The fields of the structure have the following meanings:

13251	Digest Mechanism	the mechanism type of the digest mechanism to be used (possible
13252		types can be found in [WTLS])
13253	ulMaxSizeInBits	the length (in bits) of the MACing key agreed upon during the
13254		protocol handshake phase
13255	ulKeySizeInBits	the length (in bits) of the secret key agreed upon during the
13256		handshake phase
13257	ulIVSizeInBits	the length (in bits) of the IV agreed upon during the handshake
13258		phase. If no IV is required, the length should be set to 0.
13259	ulSequenceNumber	the current sequence number used for records sent by the client
13260		and server respectively
13261	blsExport	a boolean value which indicates whether the keys have to be
13262		derives for an export version of the protocol. If this value is true (i.e.,
13263		the keys are exportable) then ulKeySizeInBits is the length of the
13264		key in bits before expansion. The length of the key after expansion
13265		is determined by the information found in the template sent along
13266		with this mechanism during a C_DeriveKey function call (either the
13267		CKA_KEY_TYPE or the CKA_VALUE_LEN attribute).
13268	RandomInfo	client's and server's random data information
13269	pReturnedKeyMaterial	points to a CK_WTLS_KEY_MAT_OUT structure which receives
13270		the handles for the keys generated and the IV

13271 **CK_WTLS_KEY_MAT_PARAMS_PTR** is a pointer to a **CK_WTLS_KEY_MAT_PARAMS**.

13272 6.41.3 Pre master secret key generation for RSA key exchange suite

13273 Pre master secret key generation for the RSA key exchange suite in WTLS denoted
13274 **CKM_WTLS_PRE_MASTER_KEY_GEN**, is a mechanism, which generates a variable length secret key.
13275 It is used to produce the pre master secret key for RSA key exchange suite used in WTLS. This
13276 mechanism returns a handle to the pre master secret key.

13277 It has one parameter, a **CK_BYTE**, which provides the client's WTLS version.

13278 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE** and **CKA_VALUE** attributes to the new
13279 key (as well as the **CKA_VALUE_LEN** attribute, if it is not supplied in the template). Other attributes may
13280 be specified in the template, or else are assigned default values.

13281 The template sent along with this mechanism during a **C_GenerateKey** call may indicate that the object
13282 class is **CKO_SECRET_KEY**, the key type is **CKK_GENERIC_SECRET**, and the **CKA_VALUE_LEN**
13283 attribute indicates the length of the pre master secret key.

13284 For this mechanism, the ulMinKeySize field of the **CK_MECHANISM_INFO** structure shall indicate 20
13285 bytes.

13286 6.41.4 Master secret key derivation

13287 Master secret derivation in WTLS, denoted **CKM_WTLS_MASTER_KEY_DERIVE**, is a mechanism used
13288 to derive a 20 byte generic secret key from variable length secret key. It is used to produce the master
13289 secret key used in WTLS from the pre master secret key. This mechanism returns the value of the client
13290 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.

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 this mechanism is only useable for key exchange suites that do not use a fixed length 20-byte pre master secret key with an embedded version number. This includes the Diffie-Hellman and Elliptic Curve Cryptography key exchange suites, but excludes the RSA key exchange suites.

6.41.6 WTLS PRF (pseudorandom function)

PRF (pseudo random function) in WTLS, denoted **CKM_WTLS_PRF**, is a mechanism used to produce a securely generated pseudo-random output of arbitrary length. The keys it uses are generic secret keys.

It has a parameter, a **CK_WTLS_PRF_PARAMS** structure, which allows for passing the mechanism type of the digest mechanism to be used, the passing of the input seed and its length, the passing of an identifying label and its length and the passing of the length of the output to the token and for receiving the output.

This mechanism produces securely generated pseudo-random output of the length specified in the parameter.

This mechanism departs from the other key derivation mechanisms in Cryptoki in not using the template sent along with this mechanism during a **C_DeriveKey** function call, which means the template shall be a NULL_PTR. For most key-derivation mechanisms, **C_DeriveKey** returns a single key handle as a result of a successful completion. However, since the **CKM_WTLS_PRF** mechanism returns the requested number of output bytes in the **CK_WTLS_PRF_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 no output will be generated.

6.41.7 Server Key and MAC derivation

Server key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_SERVER_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite 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 IV created.

It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The MACing key (server write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (server write key) is typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (server write IV) will be generated and returned if the *ulIVSizeInBits* field of the **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ulIVSizeInBits* field.

Both 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 that differ from those held by 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_SERVER_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.

6.41.8 Client key and MAC derivation

Client key, MAC and IV derivation in WTLS, denoted **CKM_WTLS_CLIENT_KEY_AND_MAC_DERIVE**, is a mechanism used to derive the appropriate cryptographic keying material used by a cipher suite 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 IV created.

It has a parameter, a **CK_WTLS_KEY_MAT_PARAMS** structure, which allows for the passing of the mechanism type of the digest mechanism to be used, random data, the characteristic of the cryptographic material for the given cipher suite, and a pointer to a structure which receives the handles and IV which were generated.

This mechanism contributes to the creation of two distinct keys and returns one IV (if an IV is requested by the caller) back to the caller. The keys are all given an object class of **CKO_SECRET_KEY**.

The MACing key (client write MAC secret) is always given a type of **CKK_GENERIC_SECRET**. It is flagged as valid for signing, verification and derivation operations.

The other key (client write key) is typed according to information found in the template sent along with this mechanism during a **C_DeriveKey** function call. By default, it is flagged as valid for encryption, decryption, and derivation operations.

An IV (client write IV) will be generated and returned if the *ullIVSizeInBits* field of the **CK_WTLS_KEY_MAT_PARAMS** field has a nonzero value. If it is generated, its length in bits will agree with the value in the *ullIVSizeInBits* field.

Both 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 that differ from those held by 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 SP800-108 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 [NIST 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_DeriveKey**. 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 195, SP800-108 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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
CKM_SP800_108_DOUBLE_PIPELINE_KDF

Data Field Types:

CK_SP800_108_ITERATION_VARIABLE
CK_SP800_108_COUNTER
CK_SP800_108_DKM_LENGTH
CK_SP800_108_BYTE_ARRAY

DKM Length Methods:

CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS

13476 CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS

13477 6.42.2 Mechanism Parameters

13478 ♦ CK_SP800_108_PRF_TYPE

13479 The CK_SP800_108_PRF_TYPE field of the mechanism parameter is used to specify the type of PRF
13480 that is to be used. It is defined as follows:

13481 typedef CK_MECHANISM_TYPE CK_SP800_108_PRF_TYPE;

13482 The CK_SP800_108_PRF_TYPE field reuses the existing mechanisms definitions. The following table
13483 lists the supported PRF types:

13484 Table 196, 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

13485

13486 ♦ CK_PRF_DATA_TYPE

13487 Each mechanism parameter contains an array of CK_PRF_DATA_PARAM structures. The
13488 CK_PRF_DATA_PARAM structure contains CK_PRF_DATA_TYPE field. The CK_PRF_DATA_TYPE
13489 field is used to identify the type of data identified by each CK_PRF_DATA_PARAM element in the array.
13490 Depending on the type of KDF used, some data field types are mandatory, some data field types are
13491 optional and some data field types are not allowed. These requirements are defined on a per-mechanism
13492 basis in the sections below. The CK_PRF_DATA_TYPE is defined as follows:

13493 typedef CK_ULONG CK_PRF_DATA_TYPE;

13494 The following table lists all of the supported 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.
CK_SP800_108_KEY_HANDLE	<p>Identifies the key handle for an object with CK_OBJECT_CLASS set to CKO_SECRET_KEY. If specified, this data type will be interpreted as an instance of CK_BYTE_ARRAY where <i>pValue</i> points to the buffer containing the CKA_VALUE attribute of the provided key handle, and <i>ulValueLen</i> is assigned the value of the CKA_VALUE_LEN attribute of the provided key handle.</p> <p>The specified key handle must have attributes settings consistent with a key that would allow it to be used as a base key for this key derivation mechanism.</p>

13496

13497 **◆ CK_PRF_DATA_PARAM**

13498 **CK_PRF_DATA_PARAM** is used to define a segment of input for the PRF. Each mechanism parameter
13499 supports an array of **CK_PRF_DATA_PARAM** structures. The **CK_PRF_DATA_PARAM** is defined as
13500 follows:

```
13501     typedef struct CK_PRF_DATA_PARAM
13502     {
13503         CK_PRF_DATA_TYPE      type;
13504         CK_VOID_PTR           pValue;
13505         CK_ULONG              ulValueLen;
13506     } CK_PRF_DATA_PARAM;
13507
13508     typedef CK_PRF_DATA_PARAM CK_PTR CK_PRF_DATA_PARAM_PTR;
13509
```

13510 The fields of the **CK_PRF_DATA_PARAM** structure have the following meaning:

13511	type	defines the type of data pointed to by pValue
13512	pValue	pointer to the data defined by type
13513	ulValueLen	size of the data pointed to by pValue

13514 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to
13515 CK_SP800_108_ITERATION_VARIABLE, then *pValue* must be set the appropriate value for the KDF's
13516 iteration variable type. For the Counter Mode KDF, *pValue* must be assigned a valid
13517 CK_SP800_108_COUNTER_FORMAT_PTR and *ulValueLen* must be set to
13518 sizeof(CK_SP800_108_COUNTER_FORMAT). For all other KDF types, *pValue must be set to*
13519 NULL_PTR and *ulValueLen* must be set to 0.

13520

13521 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_COUNTER, then
13522 *pValue* must be assigned a valid CK_SP800_108_COUNTER_FORMAT_PTR and *ulValueLen* must be
13523 set to sizeof(CK_SP800_108_COUNTER_FORMAT).

13524

13525 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_DKM_LENGTH then
13526 *pValue* must be assigned a valid CK_SP800_108_DKM_LENGTH_FORMAT_PTR and *ulValueLen* must
13527 be set to sizeof(CK_SP800_108_DKM_LENGTH_FORMAT).

13528

13529 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_BYTE_ARRAY, then
13530 *pValue* must be assigned a valid CK_BYTE_PTR value and *ulValueLen* must be set to a non-zero length.

13531

13532 If the *type* field of the **CK_PRF_DATA_PARAM** structure is set to CK_SP800_108_KEY_HANDLE, then
13533 *pValue* must be assigned a valid CK_OBJECT_HANDLE_PTR value and *ulValueLen* must be set to
13534 sizeof(CK_OBJECT_HANDLE).

13535 **◆ CK_SP800_108_COUNTER_FORMAT**

13536 **CK_SP800_108_COUNTER_FORMAT** is used to define the encoding format for a counter value. The
13537 **CK_SP800_108_COUNTER_FORMAT** is defined as follows:

```
13538     typedef struct CK_SP800_108_COUNTER_FORMAT
13539     {
13540         CK_BBOOL      bLittleEndian;
13541         CK_ULONG      ulWidthInBits;
13542     } CK_SP800_108_COUNTER_FORMAT;
13543
13544     typedef CK_SP800_108_COUNTER_FORMAT CK_PTR
13545     CK_SP800_108_COUNTER_FORMAT_PTR;
```

13547 The fields of the CK_SP800_108_COUNTER_FORMAT structure have the following meaning:

13548 bLittleEndian defines if the counter should be represented in Big Endian or Little
13549 Endian format

13550 ulWidthInBits defines the number of bits used to represent the counter value

13551 **◆ CK_SP800_108_DKM_LENGTH_METHOD**

13552 **CK_SP800_108_DKM_LENGTH_METHOD** is used to define how the DKM length value is calculated.
13553 The **CK_SP800_108_DKM_LENGTH_METHOD** type is defined as follows:

```
13554     typedef CK_ULONG CK_SP800_108_DKM_LENGTH_METHOD;
```

13555 The following table lists all of the supported DKM Length Methods:

13556 *Table 198, 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.

13557

13558 ♦ CK_SP800_108_DKM_LENGTH_FORMAT

13559 **CK_SP800_108_DKM_LENGTH_FORMAT** is used to define the encoding format for the DKM length
13560 value. The **CK_SP800_108_DKM_LENGTH_FORMAT** is defined as follows:

```

13561     typedef struct CK_SP800_108_DKM_LENGTH_FORMAT
13562     {
13563         CK_SP800_108_DKM_LENGTH_METHOD    dkmLengthMethod;
13564         CK_BBOOL                           bLittleEndian;
13565         CK_ULONG                           ulWidthInBits;
13566     } CK_SP800_108_DKM_LENGTH_FORMAT;
13567
13568     typedef CK_SP800_108_DKM_LENGTH_FORMAT CK_PTR
13569     CK_SP800_108_DKM_LENGTH_FORMAT_PTR;

```

13570

13571 The fields of the CK_SP800_108_DKM_LENGTH_FORMAT structure have the following meaning:

13572	dkmLengthMethod	defines the method used to calculate the DKM length value
13573	bLittleEndian	defines if the DKM length value should be represented in Big
13574		Endian or Little Endian format
13575	ulWidthInBits	defines the number of bits used to represent the DKM length value

13576 ♦ CK_DERIVED_KEY

13577 **CK_DERIVED_KEY** is used to define an additional key to be derived as well as provide a
13578 CK_OBJECT_HANDLE_PTR to receive the handle for the derived keys. The **CK_DERIVED_KEY** is
13579 defined as follows:

```

13580     typedef struct CK_DERIVED_KEY
13581     {
13582         CK_ATTRIBUTE_PTR    pTemplate;
13583         CK_ULONG             ulAttributeCount;
13584         CK_OBJECT_HANDLE_PTR phKey;
13585     } CK_DERIVED_KEY;
13586
13587     typedef CK_DERIVED_KEY CK_PTR CK_DERIVED_KEY_PTR;

```

13588

13589 The fields of the CK_DERIVED_KEY structure have the following meaning:

13590	pTemplate	pointer to a template that defines a key to derive
13591	ulAttributeCount	number of attributes in the template pointed to by pTemplate
13592	phKey	pointer to receive the handle for a derived key

13593 ♦ **CK_SP800_108_KDF_PARAMS, CK_SP800_108_KDF_PARAMS_PTR**

13594 **CK_SP800_108_KDF_PARAMS** is a structure that provides the parameters for the
 13595 **CKM_SP800_108_COUNTER_KDF** and **CKM_SP800_108_DOUBLE_PIPELINE_KDF** mechanisms.

```

13596
13597     typedef struct CK_SP800_108_KDF_PARAMS
13598     {
13599         CK_SP800_108_PRF_TYPE    prfType;
13600         CK_ULONG                 ulNumberOfDataParams;
13601         CK_PRF_DATA_PARAM_PTR    pDataParams;
13602         CK_ULONG                 ulAdditionalDerivedKeys;
13603         CK_DERIVED_KEY_PTR        pAdditionalDerivedKeys;
13604     } CK_SP800_108_KDF_PARAMS;
13605
13606     typedef CK_SP800_108_KDF_PARAMS CK_PTR
13607     CK_SP800_108_KDF_PARAMS_PTR;
13608
  
```

13609 The fields of the **CK_SP800_108_KDF_PARAMS** structure have the following meaning:

13610	prfType	type of PRF
13611	ulNumberOfDataParams	number of elements in the array pointed to by pDataParams
13612	pDataParams	an array of CK_PRF_DATA_PARAM structures. The array defines
13613		input parameters that are used to construct the “data” input to the
13614		PRF.
13615	ulAdditionalDerivedKeys	number of additional keys that will be derived and the number of
13616		elements in the array pointed to by pAdditionalDerivedKeys. If
13617		pAdditionalDerivedKeys is set to NULL_PTR, this parameter must
13618		be set to 0.
13619	pAdditionalDerivedKeys	an array of CK_DERIVED_KEY structures. If
13620		ulAdditionalDerivedKeys is set to 0, this parameter must be set to
13621		NULL_PTR

13622 ♦ **CK_SP800_108_FEEDBACK_KDF_PARAMS,**
 13623 **CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR**

13624 The **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure provides the parameters for the
 13625 **CKM_SP800_108_FEEDBACK_KDF** mechanism. It is defined as follows:

```

13626     typedef struct CK_SP800_108_FEEDBACK_KDF_PARAMS
13627     {
13628         CK_SP800_108_PRF_TYPE    prfType;
13629         CK_ULONG                 ulNumberOfDataParams;
13630         CK_PRF_DATA_PARAM_PTR    pDataParams;
13631         CK_ULONG                 ulIVLen;
13632         CK_BYTE_PTR              pIV;
13633         CK_ULONG                 ulAdditionalDerivedKeys;
13634         CK_DERIVED_KEY_PTR        pAdditionalDerivedKeys;
13635     } CK_SP800_108_FEEDBACK_KDF_PARAMS;
13636
  
```

13637typedef CK_SP800_108_FEEDBACK_KDF_PARAMS CK_PTR

13638CK_SP800_108_FEEDBACK_KDF_PARAMS_PTR;

13639

13640The fields of the **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure have the following meaning:

13641prfTypetype of PRF

13642ulNumberOfDataParamsnumber of elements in the array pointed to by pDataParams

13643pDataParamsan array of CK_PRF_DATA_PARAM structures. The array defines

13644input parameters that are used to construct the “data” input to the

13645PRF.

13646ulIVLenthe length in bytes of the IV. If pIV is set to NULL_PTR, this

13647parameter must be set to 0.

13648pIVan array of bytes to be used as the IV for the feedback mode KDF.

13649This parameter is optional and can be set to NULL_PTR. If ulIVLen

13650is set to 0, this parameter must be set to NULL_PTR.

13651ulAdditionalDerivedKeysnumber of additional keys that will be derived and the number of

13652elements in the array pointed to by pAdditionalDerivedKeys. If

13653pAdditionalDerivedKeys is set to NULL_PTR, this parameter must

13654be set to 0.

13655pAdditionalDerivedKeysan array of CK_DERIVED_KEY structures. If

13656ulAdditionalDerivedKeys is set to 0, this parameter must be set to

13657NULL_PTR.

13658

6.42.3 Counter Mode KDF

13659The SP800-108 Counter Mode KDF mechanism, denoted **CKM_SP800_108_COUNTER_KDF**,

13660represents the KDF defined in [NIST SP800-108] section 5.1. **CKM_SP800_108_COUNTER_KDF** is a

13661mechanism for deriving one or more symmetric keys from a symmetric base key.

13662It has a parameter, a **CK_SP800_108_KDF_PARAMS** structure.

13663The following table lists the data field types that are supported for this KDF type and their meaning:

13664

Table 199, Counter Mode data field requirements

Data Field Identifier	Description
CK_SP800_108_ITERATION_VARIABLE	This data field type is mandatory. This data field type identifies the location of the iteration variable in the constructed PRF input data. The iteration variable for this KDF type is a counter. Exact formatting of the counter value is defined by the CK_SP800_108_COUNTER_FORMAT structure.
CK_SP800_108_COUNTER	This data field type is invalid for this KDF type.
CK_SP800_108_DKM_LENGTH	This data field type is optional. This data field type identifies the location of the DKM length in the constructed PRF input data. Exact formatting of the DKM length is defined by the CK_SP800_108_DKM_LENGTH_FORMAT structure. If specified, only one instance of this type may be specified.
CK_SP800_108_BYTE_ARRAY	This data field type is optional. This data field type identifies the location and value of a byte array of data in the constructed PRF input data.

	This standard does not restrict the number of instances of this data type.
CK_SP800_108_KEY_HANDLE	<p>This data field type is optional.</p> <p>This data field type identifies the location of a symmetric key value in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p> <p>This standard does not restrict the same key handle being defined multiple times.</p>

13665

13666 [NIST SP800-108] limits the amount of derived keying material that can be produced by a Counter Mode
13667 KDF by limiting the internal loop counter to $(2^r - 1)$, where “r” is the number of bits used to represent the
13668 counter. Therefore the maximum number of bits that can be produced is $(2^r - 1)h$, where “h” is the length in
13669 bits of the output of the selected PRF.

13670 6.42.4 Feedback Mode KDF

13671 The SP800-108 Feedback Mode KDF mechanism, denoted **CKM_SP800_108_FEEDBACK_KDF**,
13672 represents the KDF defined in [NIST SP800-108] section 5.2. **CKM_SP800_108_FEEDBACK_KDF** is a
13673 mechanism for deriving one or more symmetric keys from a symmetric base key.

13674 It has a parameter, a **CK_SP800_108_FEEDBACK_KDF_PARAMS** structure.

13675 The following table lists the data field types that are supported for this KDF type and their meaning:

13676 *Table 200, Feedback 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 $K(i-1)$ in section 5.2 of [NIST 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>

	This standard does not restrict the number of instances of this data type.
CK_SP800_108_KEY_HANDLE	<p>This data field type is optional.</p> <p>This data field type identifies the location of a symmetric key value in the constructed PRF input data.</p> <p>This standard does not restrict the number of instances of this data type.</p> <p>This standard does not restrict the same key handle being defined multiple times.</p>

13677

13678 [NIST SP800-108] limits the amount of derived keying material that can be produced by a Feedback
13679 Mode KDF by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can
13680 be produced is $(2^{32}-1)h$, where “h” is the length in bits of the output of the selected PRF.

13681 6.42.5 Double Pipeline Mode KDF

13682 The SP800-108 Double Pipeline Mode KDF mechanism, denoted
13683 **CKM_SP800_108_DOUBLE_PIPELINE_KDF**, represents the KDF defined [NIST SP800-108] section
13684 5.3. **CKM_SP800_108_DOUBLE_PIPELINE_KDF** is a mechanism for deriving one or more symmetric
13685 keys from a symmetric base key.

13686 It has a parameter, a CK_SP800_108_KDF_PARAMS structure.

13687 The following table lists the data field types that are supported for this KDF type and their meaning:

13688 *Table 201, 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 [NIST 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>

	This standard does not restrict the number of instances of this data type.
CK_SP800_108_KEY_HANDLE	This data field type is optional. This data field type identifies the location of a symmetric key value in the constructed PRF input data. This standard does not restrict the number of instances of this data type. This standard does not restrict the same key handle being defined multiple times.

13689

13690

13691

13692

13693

13694

13695

13696

13697

13698

[NIST SP800-108] limits the amount of derived keying material that can be produced by a Double-Pipeline Mode KDF by limiting the internal loop counter to $(2^{32}-1)$. Therefore the maximum number of bits that can be produced is $(2^{32}-1)h$, where “h” is the length in bits of the output of the selected PRF.

The Double Pipeline KDF requires an internal IV value. The IV is constructed using the same method used to construct the PRF input data; the data/values identified by the array of **CK_PRF_DATA_PARAM** structures are concatenated in to a byte array that is used as the IV. As shown in [NIST SP800-108] section 5.3, the CK_SP800_108_ITERATION_VARIABLE and CK_SP800_108_COUNTER data field types are not included in IV construction process. All other data field types are included in the construction process.

13699

6.42.6 Deriving Additional Keys

13700

13701

13702

The KDFs defined in this section can be used to derive more than one symmetric key from the base key. The **C_DeriveKey** function accepts one CK_ATTRIBUTE_PTR to define a single derived key and one CK_OBJECT_HANDLE_PTR to receive the handle for the derived key.

13703

13704

13705

13706

13707

13708

13709

13710

13711

To derive additional keys, the mechanism parameter structure can be filled in with one or more CK_DERIVED_KEY structures. Each structure contains a CK_ATTRIBUTE_PTR to define a derived key and a CK_OBJECT_HANDLE_PTR to receive the handle for the additional derived keys. The key defined by the **C_DeriveKey** function parameters is always derived before the keys defined by the CK_DERIVED_KEY array that is part of the mechanism parameter. The additional keys that are defined by the CK_DERIVED_KEY array are derived in the order they are defined in the array. That is to say that the derived keying material produced by the KDF is processed from left to right, and bytes are assigned first to the key defined by the **C_DeriveKey** function parameters, and then bytes are assigned to the keys that are defined by the CK_DERIVED_KEY array in the order they are defined in the array.

13712

13713

13714

13715

Each internal iteration of a KDF produces a unique segment of PRF output. Sometimes, a single iteration will produce enough keying material for the key being derived. Other times, additional internal iterations are performed to produce multiple segments which are concatenated together to produce enough keying material for the derived key(s).

13716

13717

13718

13719

13720

13721

13722

When deriving multiple keys, no key can be created using part of a segment that was used for another 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_DeriveKey** 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

13723

13724

13725

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

13726 CK_SP800_108_DKM_LENGTH data field type is specified with method
 13727 CK_SP800_108_DKM_LENGTH_SUM_OF_SEGMENTS, then the DKM length value will be 768 bits.
 13728 When deriving multiple keys, if any of the keys cannot be derived for any reason, none of the keys shall
 13729 be derived. If the failure was caused by the content of a specific key's template (ie the template defined
 13730 by the content of *pTemplate*), the corresponding *phKey* value will be set to CK_INVALID_HANDLE to
 13731 identify the offending template.

13732 6.42.7 Key Derivation Attribute Rules

13733 The CKM_SP800_108_COUNTER_KDF, CKM_SP800_108_FEEDBACK_KDF and
 13734 CKM_SP800_108_DOUBLE_PIPELINE_KDF mechanisms have the following rules about key sensitivity
 13735 and extractability:

- 13736 • The **CKA_SENSITIVE** and **CKA_EXTRACTABLE** attributes in the template for the derived key(s)
 13737 can both be specified to be either CK_TRUE or CK_FALSE. If omitted, these attributes each take on
 13738 the value of these attributes as they are defined in the base key.
- 13739 • If the base key or any of the additional input keys have their **CKA_ALWAYS_SENSITIVE** attribute set
 13740 to CK_TRUE, then the derived keys will have their **CKA_ALWAYS_SENSITIVE** attribute set to the
 13741 same value as their **CKA_SENSITIVE** attribute. Otherwise the derived keys will have their
 13742 **CKA_ALWAYS_SENSITIVE** attribute set to CK_FALSE.
- 13743 • If the base key or any of the additional input keys have their **CKA_NEVER_EXTRACTABLE** attribute
 13744 set to CK_TRUE, then the derived key will have their **CKA_NEVER_EXTRACTABLE** attribute set to
 13745 the *opposite* value from their **CKA_EXTRACTABLE** attribute. Otherwise, the derived keys will have
 13746 their **CKA_NEVER_EXTRACTABLE** attribute set to CK_FALSE.

13747 6.42.8 Constructing PRF Input Data

13748 [NIST SP800-108] defines the PRF input data for each KDF at a high level using terms like “label”,
 13749 “context”, “separator”, “counter”...etc. The value, formatting and order of the input data is not strictly
 13750 defined by [NIST SP800-108], instead it is described as being defined by the “encoding scheme”.

13751 To support any encoding scheme, these mechanisms construct the PRF input data from from the array of
 13752 CK_PRF_DATA_PARAM structures in the mechanism parameter. All of the values defined by the
 13753 CK_PRF_DATA_PARAM array are concatenated in the order they are defined and passed in to the PRF
 13754 as the data parameter.

13755 6.42.8.1 Sample Counter Mode KDF

13756 [NIST SP800-108] section 5.1 outlines a sample Counter Mode KDF which defines the following PRF
 13757 input:

13758
$$\text{PRF}(K_I, [i]_2 || \text{Label} || 0x00 || \text{Context} || [L]_2)$$

13759 Section 5.1 does not define the number of bits used to represent the counter (the “r” value) or the DKM
 13760 length (the “L” value), so 16-bits is assumed for both cases. The following sample code shows how to
 13761 define this PRF input data using an array of CK_PRF_DATA_PARAM structures.

```
13762 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
13763
13764 CK_OBJECT_HANDLE hBaseKey;
13765 CK_OBJECT_HANDLE hDerivedKey;
13766 CK_OBJECT_HANDLE hAdditionalInputKey;
13767 CK_ATTRIBUTE derivedKeyTemplate = { ... };
13768
13769 CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
13770 CK_ULONG ulLabelLen = sizeof(baLabel);
13771 CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
13772 CK_ULONG ulContextLen = sizeof(baContext);
13773
```

```

13774 CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
13775 CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
13776     = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
13777
13778 CK_PRF_DATA_PARAM dataParams[] =
13779 {
13780     { CK_SP800_108_ITERATION_VARIABLE,
13781       &counterFormat, sizeof(counterFormat) },
13782     { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
13783     { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
13784     { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
13785     { CK_SP800_108_KEY_HANDLE, &hAdditionalInputKey,
13786       sizeof(hAdditionalInputKey) },
13787     { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
13788 };
13789
13790 CK_SP800_108_KDF_PARAMS kdfParams =
13791 {
13792     CKM_AES_CMACH,
13793     DIM(dataParams),
13794     &dataParams,
13795     0,          /* no addition derived keys */
13796     NULL        /* no addition derived keys */
13797 };
13798
13799 CK_MECHANISM = mechanism
13800 {
13801     CKM_SP800_108_COUNTER_KDF,
13802     &kdfParams,
13803     sizeof(kdfParams)
13804 };
13805
13806 hBaseKey = GetBaseKeyHandle(.....);
13807
13808 hAdditionalInputKey = GetAdditionalInputKeyHandle(.....);
13809
13810 rv = C_DeriveKey(
13811     hSession,
13812     &mechanism,
13813     hBaseKey,
13814     &derivedKeyTemplate,
13815     DIM(derivedKeyTemplate),
13816     &hDerivedKey);

```

13817 6.42.8.2 Sample SCP03 Counter Mode KDF

13818 The SCP03 standard defines a variation of a counter mode KDF which defines the following PRF input:

13819 **PRF** (*K_i*, *Label* || 0x00 || [*L*]₂ || [*i*]₂ || *Context*)

13820 SCP03 defines the number of bits used to represent the counter (the “r” value) and number of bits used to
13821 represent the DKM length (the “L” value) as 16-bits. The following sample code shows how to define this
13822 PRF input data using an array of CK_PRF_DATA_PARAM structures.

```

13823 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
13824
13825 CK_OBJECT_HANDLE hBaseKey;
13826 CK_OBJECT_HANDLE hDerivedKey;
13827 CK_ATTRIBUTE derivedKeyTemplate = { ... };
13828

```

```

13829 CK_BYTE baLabel[] = {0xde, 0xad, 0xbe , 0xef};
13830 CK_ULONG ulLabelLen = sizeof(baLabel);
13831 CK_BYTE baContext[] = {0xfe, 0xed, 0xbe , 0xef};
13832 CK_ULONG ulContextLen = sizeof(baContext);
13833
13834 CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
13835 CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
13836     = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
13837
13838 CK_PRF_DATA_PARAM dataParams[] =
13839 {
13840     { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
13841     { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
13842     { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) },
13843     { CK_SP800_108_ITERATION_VARIABLE,
13844         &counterFormat, sizeof(counterFormat) },
13845     { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen }
13846 };
13847
13848 CK_SP800_108_KDF_PARAMS kdfParams =
13849 {
13850     CKM_AES_CMACH,
13851     DIM(dataParams),
13852     &dataParams,
13853     0, /* no addition derived keys */
13854     NULL /* no addition derived keys */
13855 };
13856
13857 CK_MECHANISM = mechanism
13858 {
13859     CKM_SP800_108_COUNTER_KDF,
13860     &kdfParams,
13861     sizeof(kdfParams)
13862 };
13863
13864 hBaseKey = GetBaseKeyHandle(.....);
13865
13866 rv = C_DeriveKey(
13867     hSession,
13868     &mechanism,
13869     hBaseKey,
13870     &derivedKeyTemplate,
13871     DIM(derivedKeyTemplate),
13872     &hDerivedKey);

```

13873 6.42.8.3 Sample Feedback Mode KDF

13874 [NIST SP800-108] section 5.2 outlines a sample Feedback Mode KDF which defines the following PRF
 13875 input:

13876
$$\text{PRF}(K_i, K(i-1) \parallel [i]_2 \parallel \text{Label} \parallel 0x00 \parallel \text{Context} \parallel [L]_2)$$

13877 Section 5.2 does not define the number of bits used to represent the counter (the “r” value) or the DKM
 13878 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional and
 13879 is included in this example. The following sample code shows how to define this PRF input data using an
 13880 array of CK_PRF_DATA_PARAM structures.

```

13881 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
13882

```

```

13883 CK_OBJECT_HANDLE hBaseKey;
13884 CK_OBJECT_HANDLE hDerivedKey;
13885 CK_ATTRIBUTE derivedKeyTemplate = { ... };
13886
13887 CK_BYTE baFeedbackIV[] = {0x01, 0x02, 0x03, 0x04};
13888 CK_ULONG ulFeedbackIVLen = sizeof(baFeedbackIV);
13889 CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
13890 CK_ULONG ulLabelLen = sizeof(baLabel);
13891 CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
13892 CK_ULONG ulContextLen = sizeof(baContext);
13893
13894 CK_SP800_108_COUNTER_FORMAT counterFormat = {0, 16};
13895 CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
13896     = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
13897
13898 CK_PRF_DATA_PARAM dataParams[] =
13899 {
13900     { CK_SP800_108_ITERATION_VARIABLE,
13901       &counterFormat, sizeof(counterFormat) },
13902     { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
13903     { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
13904     { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
13905     { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
13906 };
13907
13908 CK_SP800_108_FEEDBACK_KDF_PARAMS kdfParams =
13909 {
13910     CKM_AES_CMAC,
13911     DIM(dataParams),
13912     &dataParams,
13913     ulFeedbackIVLen,
13914     baFeedbackIV,
13915     0,          /* no addition derived keys */
13916     NULL        /* no addition derived keys */
13917 };
13918
13919 CK_MECHANISM = mechanism
13920 {
13921     CKM_SP800_108_FEEDBACK_KDF,
13922     &kdfParams,
13923     sizeof(kdfParams)
13924 };
13925
13926 hBaseKey = GetBaseKeyHandle(.....);
13927
13928 rv = C_DeriveKey(
13929     hSession,
13930     &mechanism,
13931     hBaseKey,
13932     &derivedKeyTemplate,
13933     DIM(derivedKeyTemplate),
13934     &hDerivedKey);

```

13935 6.42.8.4 Sample Double-Pipeline Mode KDF

13936 [NIST SP800-108] section 5.3 outlines a sample Double-Pipeline Mode KDF which defines the two
 13937 following PRF inputs:

13938 PRF (K_i , $A(i-1)$)

```

13939         PRF(KI, K(i-1) || [i]2 || Label || 0x00 || Context || [L]2)
13940
13941 Section 5.3 does not define the number of bits used to represent the counter (the “r” value) or the DKM
13942 length (the “L” value), so 16-bits is assumed for both cases. The counter is defined as being optional so it
13943 is left out in this example. The following sample code shows how to define this PRF input data using an
13944 array of CK_PRF_DATA_PARAM structures.
13945
13946 #define DIM(a) (sizeof((a))/sizeof((a)[0]))
13947
13948 CK_OBJECT_HANDLE hBaseKey;
13949 CK_OBJECT_HANDLE hDerivedKey;
13950 CK_ATTRIBUTE derivedKeyTemplate = { ... };
13951
13952 CK_BYTE baLabel[] = {0xde, 0xad, 0xbe, 0xef};
13953 CK_ULONG ulLabelLen = sizeof(baLabel);
13954 CK_BYTE baContext[] = {0xfe, 0xed, 0xbe, 0xef};
13955 CK_ULONG ulContextLen = sizeof(baContext);
13956
13957 CK_SP800_108_DKM_LENGTH_FORMAT dkmFormat
13958     = {CK_SP800_108_DKM_LENGTH_SUM_OF_KEYS, 0, 16};
13959
13960 CK_PRF_DATA_PARAM dataParams[] =
13961 {
13962     { CK_SP800_108_BYTE_ARRAY, baLabel, ulLabelLen },
13963     { CK_SP800_108_BYTE_ARRAY, {0x00}, 1 },
13964     { CK_SP800_108_BYTE_ARRAY, baContext, ulContextLen },
13965     { CK_SP800_108_DKM_LENGTH, dkmFormat, sizeof(dkmFormat) }
13966 };
13967
13968 CK_SP800_108_KDF_PARAMS kdfParams =
13969 {
13970     CKM_AES_CMAC,
13971     DIM(dataParams),
13972     &dataParams,
13973     0, /* no addition derived keys */
13974     NULL /* no addition derived keys */
13975 };
13976
13977 CK_MECHANISM = mechanism
13978 {
13979     CKM_SP800_108_DOUBLE_PIPELINE_KDF,
13980     &kdfParams,
13981     sizeof(kdfParams)
13982 };
13983
13984 hBaseKey = GetBaseKeyHandle(.....);
13985
13986 rv = C_DeriveKey(
13987     hSession,
13988     &mechanism,
13989     hBaseKey,
13990     &derivedKeyTemplate,
13991     DIM(derivedKeyTemplate),
13992     &hDerivedKey);

```

13991 6.43 Miscellaneous simple key derivation mechanisms

13992 *Table 202, Miscellaneous simple key derivation Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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							✓	
CKM_PUB_KEY_FROM_PRIV_KEY							✓	

6.43.1 Definitions

Mechanisms:

CKM_CONCATENATE_BASE_AND_DATA
CKM_CONCATENATE_DATA_AND_BASE
CKM_XOR_BASE_AND_DATA
CKM_EXTRACT_KEY_FROM_KEY
CKM_CONCATENATE_BASE_AND_KEY
CKM_PUB_KEY_FROM_PRIV_KEY

6.43.2 Parameters for miscellaneous simple key derivation mechanisms

♦ CK_KEY_DERIVATION_STRING_DATA; CK_KEY_DERIVATION_STRING_DATA_PTR

CK_KEY_DERIVATION_STRING_DATA provides the parameters for the **CKM_CONCATENATE_BASE_AND_DATA**, **CKM_CONCATENATE_DATA_AND_BASE**, and **CKM_XOR_BASE_AND_DATA** mechanisms. It is defined as follows:

```
typedef struct CK_KEY_DERIVATION_STRING_DATA {
    CK_BYTE_PTR pData;
    CK_ULONG ulLen;
} CK_KEY_DERIVATION_STRING_DATA;
```

The fields of the structure have the following meanings:

pData pointer to the byte string
ulLen length of the byte string

CK_KEY_DERIVATION_STRING_DATA_PTR is a pointer to a **CK_KEY_DERIVATION_STRING_DATA**.

♦ CK_EXTRACT_PARAMS; CK_EXTRACT_PARAMS_PTR

CK_EXTRACT_PARAMS provides the parameter to the **CKM_EXTRACT_KEY_FROM_KEY** mechanism. It specifies which bit of the base key should be used as the first bit of the derived key. It is defined as follows:

```
typedef CK_ULONG CK_EXTRACT_PARAMS;
```

14022

14023 **CK_EXTRACT_PARAMS_PTR** is a pointer to a **CK_EXTRACT_PARAMS**.

14024 **6.43.3 Concatenation of a base key and another key**

14025 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_KEY**, derives a secret key from the
14026 concatenation of two existing secret keys. The two keys are specified by handles; the values of the keys
14027 specified are concatenated together in a buffer.

14028 This mechanism takes a parameter, a **CK_OBJECT_HANDLE**. This handle produces the key value
14029 information which is appended to the end of the base key's value information (the base key is the key
14030 whose handle is supplied as an argument to **C_DeriveKey**).

14031 For example, if the value of the base key is 0x01234567, and the value of the other key is 0x89ABCDEF,
14032 then the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 14033 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
14034 generic secret key. Its length will be equal to the sum of the lengths of the values of the two original
14035 keys.
- 14036 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
14037 will be a generic secret key of the specified length.
- 14038 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
14039 length. If it does, then the key produced by this mechanism will be of the type specified in the
14040 template. If it doesn't, an error will be returned.
- 14041 • If both a key type and a length are provided in the template, the length must be compatible with that
14042 key type. The key produced by this mechanism will be of the specified type and length.

14043 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
14044 properly.

14045 If the requested type of key requires more bytes than are available by concatenating the two original keys'
14046 values, an error is generated.

14047 This mechanism has the following rules about key sensitivity and extractability:

- 14048 • If either of the two original keys has its **CKA_SENSITIVE** attribute set to **CK_TRUE**, so does the
14049 derived key. If not, then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied
14050 template or from a default value.
- 14051 • Similarly, if either of the two original keys has its **CKA_EXTRACTABLE** attribute set to **CK_FALSE**,
14052 so does the derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either
14053 from the supplied template or from a default value.
- 14054 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to **CK_TRUE** if and only if both of the
14055 original keys have their **CKA_ALWAYS_SENSITIVE** attributes set to **CK_TRUE**.
- 14056 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to **CK_TRUE** if and only if
14057 both of the original keys have their **CKA_NEVER_EXTRACTABLE** attributes set to **CK_TRUE**.

14058 **6.43.4 Concatenation of a base key and data**

14059 This mechanism, denoted **CKM_CONCATENATE_BASE_AND_DATA**, derives a secret key by
14060 concatenating data onto the end of a specified secret key.

14061 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
14062 specifies the length and value of the data which will be appended to the base key to derive another key.

14063 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
14064 the value of the derived key will be taken from a buffer containing the string 0x0123456789ABCDEF.

- 14065 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
14066 generic secret key. Its length will be equal to the sum of the lengths of the value of the original key
14067 and the data.

- 14068 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
14069 will be a generic secret key of the specified length.
- 14070 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
14071 length. If it does, then the key produced by this mechanism will be of the type specified in the
14072 template. If it doesn't, an error will be returned.
- 14073 • If both a key type and a length are provided in the template, the length must be compatible with that
14074 key type. The key produced by this mechanism will be of the specified type and length.
- 14075 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
14076 properly.
- 14077 If the requested type of key requires more bytes than are available by concatenating the original key's
14078 value and the data, an error is generated.
- 14079 This mechanism has the following rules about key sensitivity and extractability:
- 14080 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
14081 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
14082 default value.
- 14083 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
14084 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
14085 supplied template or from a default value.
- 14086 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
14087 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 14088 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
14089 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

14090 6.43.5 Concatenation of data and a base key

- 14091 This mechanism, denoted **CKM_CONCATENATE_DATA_AND_BASE**, derives a secret key by
14092 prepending data to the start of a specified secret key.
- 14093 This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which
14094 specifies the length and value of the data which will be prepended to the base key to derive another key.
- 14095 For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then
14096 the value of the derived key will be taken from a buffer containing the string 0x89ABCDEF01234567.
- 14097 • If no length or key type is provided in the template, then the key produced by this mechanism will be a
14098 generic secret key. Its length will be equal to the sum of the lengths of the data and the value of the
14099 original key.
- 14100 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
14101 will be a generic secret key of the specified length.
- 14102 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
14103 length. If it does, then the key produced by this mechanism will be of the type specified in the
14104 template. If it doesn't, an error will be returned.
- 14105 • If both a key type and a length are provided in the template, the length must be compatible with that
14106 key type. The key produced by this mechanism will be of the specified type and length.
- 14107 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
14108 properly.
- 14109 If the requested type of key requires more bytes than are available by concatenating the data and the
14110 original key's value, an error is generated.
- 14111 This mechanism has the following rules about key sensitivity and extractability:
- 14112 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
14113 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
14114 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.

6.43.6 XORing of a key and data

XORing key derivation, denoted **CKM_XOR_BASE_AND_DATA**, is a mechanism which provides the capability of deriving a secret key by performing a bit XORing of a key pointed to by a base key handle and some data.

This mechanism takes a parameter, a **CK_KEY_DERIVATION_STRING_DATA** structure, which specifies the data with which to XOR the original key's value.

For example, if the value of the base key is 0x01234567, and the value of the data is 0x89ABCDEF, then the value of the derived key will be taken from a buffer containing the string 0x88888888.

- If no length or key type is provided in the template, then the key produced by this mechanism will be a generic secret key. Its length will be equal to the minimum of the lengths of the data and the value of the original key.
- 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 are available by taking the shorter of the data and the original key's value, 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.

6.43.7 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.

14160 We give an example of how this mechanism works. Suppose a token has a secret key with the 4-byte
 14161 value 0x329F84A9. We will derive a 2-byte secret key from this key, starting at bit position 21 (i.e., the
 14162 value of the parameter to the **CKM_EXTRACT_KEY_FROM_KEY** mechanism is 21).

- 14163 1. We write the key's value in binary: 0011 0010 1001 1111 1000 0100 1010 1001. We regard this
 14164 binary string as holding the 32 bits of the key, labeled as b0, b1, ..., b31.
- 14165 2. We then extract 16 consecutive bits (i.e., 2 bytes) from this binary string, starting at bit b21. We obtain
 14166 the binary string 1001 0101 0010 0110.
- 14167 3. The value of the new key is thus 0x9526.

14168 Note that when constructing the value of the derived key, it is permissible to wrap around the end of the
 14169 binary string representing the original key's value.

14170 If the original key used in this process is sensitive, then the derived key must also be sensitive for the
 14171 derivation to succeed.

- 14172 • If no length or key type is provided in the template, then an error will be returned.
- 14173 • If no key type is provided in the template, but a length is, then the key produced by this mechanism
 14174 will be a generic secret key of the specified length.
- 14175 • If no length is provided in the template, but a key type is, then that key type must have a well-defined
 14176 length. If it does, then the key produced by this mechanism will be of the type specified in the
 14177 template. If it doesn't, an error will be returned.
- 14178 • If both a key type and a length are provided in the template, the length must be compatible with that
 14179 key type. The key produced by this mechanism will be of the specified type and length.

14180 If a DES, DES2, DES3, or CDMF key is derived with this mechanism, the parity bits of the key will be set
 14181 properly.

14182 If the requested type of key requires more bytes than the original key has, an error is generated.

14183 This mechanism has the following rules about key sensitivity and extractability:

- 14184 • If the base key has its **CKA_SENSITIVE** attribute set to CK_TRUE, so does the derived key. If not,
 14185 then the derived key's **CKA_SENSITIVE** attribute is set either from the supplied template or from a
 14186 default value.
- 14187 • Similarly, if the base key has its **CKA_EXTRACTABLE** attribute set to CK_FALSE, so does the
 14188 derived key. If not, then the derived key's **CKA_EXTRACTABLE** attribute is set either from the
 14189 supplied template or from a default value.
- 14190 • The derived key's **CKA_ALWAYS_SENSITIVE** attribute is set to CK_TRUE if and only if the base
 14191 key has its **CKA_ALWAYS_SENSITIVE** attribute set to CK_TRUE.
- 14192 • Similarly, the derived key's **CKA_NEVER_EXTRACTABLE** attribute is set to CK_TRUE if and only if
 14193 the base key has its **CKA_NEVER_EXTRACTABLE** attribute set to CK_TRUE.

14194 6.43.8 Public key from private key

14195 Public key from private key, denoted **CKM_PUB_KEY_FROM_PRIV_KEY**, is a mechanism which
 14196 creates a matching public key from an existing private key. This mechanism takes no parameters. The
 14197 public key matches the private key's type, and all the key specific parameters are generated by the
 14198 mechanism. If the following parameters are not set in the template, they will default as follows:

- 14199 1. CKA_TOKEN – CK_FALSE
- 14200 2. CKA_PRIVATE – CK_FALSE
- 14201 3. CKA_MODIFIABLE – CK_TRUE
- 14202 4. CKA_LOCAL – CK_FALSE
- 14203 5. CKA_SENSITIVE – CK_FALSE
- 14204 6. CKA_ALWAYS_SENSITIVE – CK_FALSE
- 14205 7. CKA_EXTRACTABLE – CK_TRUE
- 14206 8. CKA_NEVER_EXTRACTABLE – CK_FALSE
- 14207 9. CKA_COPYABLE – CK_TRUE
- 14208 10. CKA_DESTROYABLE – CK_FALSE

- 14209
11. CKA_LABEL – NULL
- 14210
12. CKA_WRAP_TEMPLATE - empty
- 14211
13. CKA_ENCRYPT – to CKA_DECRYPT of the private key
- 14212
14. CKA_VERIFY – to CKA_SIGN of the private key
- 14213
15. CKA_VERIFY_RECOVER – to CKA_SIGN_RECOVER of the private key
- 14214
16. CKA_WRAP – to CKA_UNWRAP of the private key
- 14215
17. CKA_DERIVE – to CKA_DERIVE of the private key
- 14216
18. CKA_ID – copy from private key
- 14217
19. CKA_START_DATE – copy from private key
- 14218
20. CKA_END_DATE – copy from private key
- 14219
21. CKA_SUBJECT – copy from private key
- 14220
22. CKA_PUBLIC_KEY_INFO – copy from private key
- 14221
23. CKA_KEY_GEN_MECHANISM – copy from private key
- 14222
24. CKA_ALLOWED_MECHANISMS – copy from private key
- 14223
25. CKA_TRUSTED – copy from private key
- 14224
- This mechanism should be supported for all private key types supported by the token.

14225

6.44 CMS

14226

Table 203, CMS Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_CMS_SIG		✓	✓					

14227

6.44.1 Definitions

- 14228
- Mechanisms:
- 14229
- CKM_CMS_SIG

14230

6.44.2 CMS Signature Mechanism Objects

- 14231
- These objects provide information relating to the **CKM_CMS_SIG** mechanism. **CKM_CMS_SIG**
- 14232
- mechanism object attributes represent information about supported CMS signature attributes in the token.
- 14233
- They are only present on tokens supporting the **CKM_CMS_SIG** mechanism, but must be present on
- 14234
- those tokens.

14235

Table 204, 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

- 14236
- The contents of each byte array will be a DER-encoded list of CMS **Attributes** with optional accompanying
- 14237
- values. Any attributes in the list shall be identified with its object identifier, and any values shall be DER-
- 14238
- encoded. The list of attributes is defined in ASN.1 as:

```

14239     Attributes ::= SET SIZE (1..MAX) OF Attribute
14240     Attribute ::= SEQUENCE {
14241         attrType      OBJECT IDENTIFIER,
14242         attrValues SET OF ANY DEFINED BY OBJECT IDENTIFIER
14243             OPTIONAL
14244     }

```

14245 The client may not set any of the attributes.

14246 6.44.3 CMS mechanism parameters

14247 • CK_CMS_SIG_PARAMS, CK_CMS_SIG_PARAMS_PTR

14248 **CK_CMS_SIG_PARAMS** is a structure that provides the parameters to the **CKM_CMS_SIG** mechanism.
14249 It is defined as follows:

```

14250     typedef struct CK_CMS_SIG_PARAMS {
14251         CK_OBJECT_HANDLE      certificateHandle;
14252         CK_MECHANISM_PTR      pSigningMechanism;
14253         CK_MECHANISM_PTR      pDigestMechanism;
14254         CK_UTF8CHAR_PTR       pContentType;
14255         CK_BYTE_PTR           pRequestedAttributes;
14256         CK_ULONG              ulRequestedAttributesLen;
14257         CK_BYTE_PTR           pRequiredAttributes;
14258         CK_ULONG              ulRequiredAttributesLen;
14259     } CK_CMS_SIG_PARAMS;

```

14260

14261 The fields of the structure have the following meanings:

14262	certificateHandle	Object handle for a certificate associated with the signing key. The
14263		token may use information from this certificate to identify the signer
14264		in the SignerInfo result value. CertificateHandle may be NULL_PTR
14265		if the certificate is not available as a PKCS #11 object or if the
14266		calling application leaves the choice of certificate completely to the
14267		token.
14268	pSigningMechanism	Mechanism to use when signing a constructed CMS
14269		SignedAttributes value. E.g. CKM_SHA1_RSA_PKCS .
14270	pDigestMechanism	Mechanism to use when digesting the data. Value shall be
14271		NULL_PTR when the digest mechanism to use follows from the
14272		pSigningMechanism parameter.
14273	pContentType	NULL-terminated string indicating complete MIME Content-type of
14274		message to be signed; or the value NULL_PTR if the message is a
14275		MIME object (which the token can parse to determine its MIME
14276		Content-type if required). Use the value "application/octet-stream" if
14277		the MIME type for the message is unknown or undefined. Note that
14278		the pContentType string shall conform to the syntax specified in
14279		[RFC 2045], i.e. any parameters needed for correct presentation of
14280		the content by the token (such as, for example, a non-default
14281		"charset") must be present. The token must follow rules and
14282		procedures defined in [RFC 2045] when presenting the content.
14283	pRequestedAttributes	Pointer to DER-encoded list of CMS Attributes the caller requests to
14284		be included in the signed attributes. Token may freely ignore this list
14285		or modify any supplied values.

14286	<code>ulRequestedAttributesLen</code>	Length in bytes of the value pointed to by <code>pRequestedAttributes</code>
14287	<code>pRequiredAttributes</code>	Pointer to DER-encoded list of CMS Attributes (with accompanying values) required to be included in the resulting signed attributes.
14288		Token must not modify any supplied values. If the token does not
14289		support one or more of the attributes, or does not accept provided
14290		values, the signature operation will fail. The token will use its own
14291		default attributes when signing if both the <code>pRequestedAttributes</code> and
14292		<code>pRequiredAttributes</code> field are set to <code>NULL_PTR</code> .
14293		
14294	<code>ulRequiredAttributesLen</code>	Length in bytes, of the value pointed to by <code>pRequiredAttributes</code> .

14295 6.44.4 CMS signatures

14296 The CMS mechanism, denoted **CKM_CMS_SIG**, is a multi-purpose mechanism based on the structures
14297 defined in [PKCS #7] and [RFC 5652]. It supports single- or multiple-part signatures with and without
14298 message recovery. The mechanism is intended for use with, e.g., PTDs (see [MeT-PTD]) or other
14299 capable tokens. The token will construct a CMS **SignedAttributes** value and compute a signature on this
14300 value. The content of the **SignedAttributes** value is decided by the token, however the caller can suggest
14301 some attributes in the parameter `pRequestedAttributes`. The caller can also require some attributes to be
14302 present through the parameters `pRequiredAttributes`. The signature is computed in accordance with the
14303 parameter `pSigningMechanism`.

14304 When this mechanism is used in successful calls to **C_Sign** or **C_SignFinal**, the `pSignature` return value
14305 will point to a DER-encoded value of type **SignerInfo**. **SignerInfo** is defined in ASN.1 as follows (for a
14306 complete definition of all fields and types, see [RFC 5652]):

```

14307     SignerInfo ::= SEQUENCE {
14308         version CMSVersion,
14309         sid SignerIdentifier,
14310         digestAlgorithm DigestAlgorithmIdentifier,
14311         signedAttrs [0] IMPLICIT SignedAttributes OPTIONAL,
14312         signatureAlgorithm SignatureAlgorithmIdentifier,
14313         signature SignatureValue,
14314         unsignedAttrs [1] IMPLICIT UnsignedAttributes
14315         OPTIONAL }

```

14316 The `certificateHandle` parameter, when set, helps the token populate the `sid` field of the **SignerInfo** value.
14317 If `certificateHandle` is `NULL_PTR` the choice of a suitable certificate reference in the **SignerInfo** result
14318 value is left to the token (the token could, e.g., interact with the user).

14319 This mechanism shall not be used in calls to **C_Verify** or **C_VerifyFinal** (use the `pSigningMechanism`
14320 mechanism instead).

14321 For the `pRequiredAttributes` field, the token may have to interact with the user to find out whether to
14322 accept a proposed value or not. The token should never accept any proposed attribute values without
14323 some kind of confirmation from its owner (but this could be through, e.g., configuration or policy settings
14324 and not direct interaction). If a user rejects proposed values, or the signature request as such, the value
14325 **CKR_FUNCTION_REJECTED** shall be returned.

14326 When possible, applications should use the **CKM_CMS_SIG** mechanism when generating CMS-
14327 compatible signatures rather than lower-level mechanisms such as **CKM_SHA1_RSA_PKCS**. This is
14328 especially true when the signatures are to be made on content that the token is able to present to a user.
14329 Exceptions may include those cases where the token does not support a particular signing attribute. Note
14330 however that the token may refuse usage of a particular signature key unless the content to be signed is
14331 known (i.e. the **CKM_CMS_SIG** mechanism is used).

14332 When a token does not have presentation capabilities, the PKCS #11-aware application may avoid
14333 sending the whole message to the token by electing to use a suitable signature mechanism (e.g.
14334 **CKM_RSA_PKCS**) as the `pSigningMechanism` value in the **CK_CMS_SIG_PARAMS** structure, and
14335 digesting the message itself before passing it to the token.

14336 PKCS #11-aware applications making use of tokens with presentation capabilities, should attempt to
14337 provide messages to be signed by the token in a format possible for the token to present to the user.
14338 Tokens that receive multipart MIME-messages for which only certain parts are possible to present may
14339 fail the signature operation with a return value of **CKR_DATA_INVALID**, but may also choose to add a
14340 signing attribute indicating which parts of the message were possible to present.

14341 **6.45 Blowfish**

14342 Blowfish, a secret-key block cipher. It is a Feistel network, iterating a simple encryption function 16 times.
14343 The block size is 64 bits, and the key can be any length up to 448 bits. Although there is a complex
14344 initialization phase required before any encryption can take place, the actual encryption of data is very
14345 efficient on large microprocessors. See [BLOWFISH] for details.

14346

14347 *Table 205, Blowfish Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_BLOWFISH_KEY_GEN					✓			
CKM_BLOWFISH_CBC	✓					✓		
CKM_BLOWFISH_CBC_PAD	✓					✓		

14348 **6.45.1 Definitions**

14349 This section defines the key type “**CKK_BLOWFISH**” for type CK_KEY_TYPE as used in the
14350 **CKA_KEY_TYPE** attribute of key objects.

14351 Mechanisms:

- 14352 CKM_BLOWFISH_KEY_GEN
- 14353 CKM_BLOWFISH_CBC
- 14354 CKM_BLOWFISH_CBC_PAD

14355 **6.45.2 BLOWFISH secret key objects**

14356 Blowfish secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_BLOWFISH**) hold Blowfish
14357 keys. The following table defines the Blowfish secret key object attributes, in addition to the common
14358 attributes defined for this object class:

14359 *Table 206, 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

14360 ¹Refer to Table 13 for footnotes

14361 The following is a sample template for creating an Blowfish secret key object:

```
14362 CK_OBJECT_CLASS class = CKO_SECRET_KEY;  
14363 CK_KEY_TYPE keyType = CKK_BLOWFISH;  
14364 CK_UTF8CHAR label[] = "A blowfish secret key object";  
14365 CK_BYTE value[16] = {...};
```

```
14366 CK_BBOOL true = CK_TRUE;
14367 CK_ATTRIBUTE template[] = {
14368     {CKA_CLASS, &class, sizeof(class)},
14369     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14370     {CKA_TOKEN, &true, sizeof(true)},
14371     {CKA_LABEL, label, sizeof(label)-1},
14372     {CKA_ENCRYPT, &true, sizeof(true)},
14373     {CKA_VALUE, value, sizeof(value)}
14374 };
```

14375 **6.45.3 Blowfish key generation**

14376 The Blowfish key generation mechanism, denoted **CKM_BLOWFISH_KEY_GEN**, is a key generation
14377 mechanism Blowfish.

14378 It does not have a parameter.

14379 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN**
14380 attribute of the template for the key.

14381 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
14382 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
14383 supports) may be specified in the template for the key, or else are assigned default initial values.

14384 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14385 specify the supported range of key sizes in bytes.

14386 **6.45.4 Blowfish-CBC**

14387 Blowfish-CBC, denoted **CKM_BLOWFISH_CBC**, is a mechanism for single- and multiple-part encryption
14388 and decryption; key wrapping; and key unwrapping.

14389 It has a parameter, a 8-byte initialization vector.

14390 This mechanism can wrap and unwrap any secret key. For wrapping, the mechanism encrypts the value
14391 of the **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size
14392 minus one null bytes so that the resulting length is a multiple of the block size. The output data is the
14393 same length as the padded input data. It does not wrap the key type, key length, or any other information
14394 about the key; the application must convey these separately.

14395 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
14396 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
14397 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
14398 attribute of the new key; other attributes required by the key type must be specified in the template.

14399 Constraints on key types and the length of data are summarized in the following table:

14400 *Table 207, 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

14401 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14402 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.

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

Table 208, 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

6.46 Twofish

Twofish is a secret key block cipher. See [TWOFISH] for details.

Table 209, Twofish Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_TWOFISH_KEY_GEN					✓			
CKM_TWOFISH_CBC	✓					✓		
CKM_TWOFISH_CBC_PAD	✓					✓		

6.46.1 Definitions

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

Mechanisms:

CKM_TWOFISH_KEY_GEN

CKM_TWOFISH_CBC

CKM_TWOFISH_CBC_PAD

14427

14428 **6.46.2 Twofish secret key objects**

14429
14430
14431

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

14432 *Table 210, 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

14433 Refer to Table 13 for footnotes

14434 The following is a sample template for creating an TWOFISH secret key object:

```
14435 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
14436 CK_KEY_TYPE keyType = CKK_TWOFISH;
14437 CK_UTF8CHAR label[] = "A twofish secret key object";
14438 CK_BYTE value[16] = {...};
14439 CK_BBOOL true = CK_TRUE;
14440 CK_ATTRIBUTE template[] = {
14441     {CKA_CLASS, &class, sizeof(class)},
14442     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14443     {CKA_TOKEN, &true, sizeof(true)},
14444     {CKA_LABEL, label, sizeof(label)-1},
14445     {CKA_ENCRYPT, &true, sizeof(true)},
14446     {CKA_VALUE, value, sizeof(value)}
14447 };
```

14448 **6.46.3 Twofish key generation**

14449 The Twofish key generation mechanism, denoted **CKM_TWOFISH_KEY_GEN**, is a key generation
14450 mechanism Twofish.

14451 It does not have a parameter.

14452 The mechanism generates Blowfish keys with a particular length, as specified in the **CKA_VALUE_LEN**
14453 attribute of the template for the key.

14454 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
14455 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
14456 supports) may be specified in the template for the key, or else are assigned default initial values.

14457 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14458 specify the supported range of key sizes, in bytes.

14459 **6.46.4 Twofish -CBC**

14460 Twofish-CBC, denoted **CKM_TWOFISH_CBC**, is a mechanism for single- and multiple-part encryption
14461 and decryption; key wrapping; and key unwrapping.

14462 It has a parameter, a 16-byte initialization vector.

6.46.5 Twofish-CBC with PKCS padding

Twofish-CBC-PAD, denoted **CKM_TWOFISH_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 16-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.

6.47 CAMELLIA

Camellia is a block cipher with 128-bit block size and 128-, 192-, and 256-bit keys, similar to AES.

Camellia is described e.g. in IETF [RFC 3713].

Table 211, *Camellia Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_CAMELLIA_KEY_GEN					✓			
CKM_CAMELLIA_ECB	✓					✓		
CKM_CAMELLIA_CBC	✓					✓		
CKM_CAMELLIA_CBC_PAD	✓					✓		
CKM_CAMELLIA_MAC_GENERAL		✓						
CKM_CAMELLIA_MAC		✓						
CKM_CAMELLIA_ECB_ENCRYPT_DATA							✓	
CKM_CAMELLIA_CBC_ENCRYPT_DATA							✓	

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:

14489 Table 212, 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}	CK_ULONG	Length in bytes of key value

14490 Refer to Table 13 for footnotes

14491 The following is a sample template for creating a Camellia secret key object:

```
14492 CK_OBJECT_CLASS class = CKO_SECRET_KEY;
14493 CK_KEY_TYPE keyType = CKK_CAMELLIA;
14494 CK_UTF8CHAR label[] = "A Camellia secret key object";
14495 CK_BYTE value[] = {...};
14496 CK_BBOOL true = CK_TRUE;
14497 CK_ATTRIBUTE template[] = {
14498     {CKA_CLASS, &class, sizeof(class)},
14499     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14500     {CKA_TOKEN, &true, sizeof(true)},
14501     {CKA_LABEL, label, sizeof(label)-1},
14502     {CKA_ENCRYPT, &true, sizeof(true)},
14503     {CKA_VALUE, value, sizeof(value)}
14504 };
```

14505 **6.47.3 Camellia key generation**

14506 The Camellia key generation mechanism, denoted **CKM_CAMELLIA_KEY_GEN**, is a key generation
14507 mechanism for Camellia.

14508 It does not have a parameter.

14509 The mechanism generates Camellia keys with a particular length in bytes, as specified in the
14510 **CKA_VALUE_LEN** attribute of the template for the key.

14511 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
14512 key. Other attributes supported by the Camellia key type (specifically, the flags indicating which functions
14513 the key supports) may be specified in the template for the key, or else are assigned default initial values.

14514 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14515 specify the supported range of Camellia key sizes, in bytes.

14516 **6.47.4 Camellia-ECB**

14517 Camellia-ECB, denoted **CKM_CAMELLIA_ECB**, is a mechanism for single- and multiple-part encryption
14518 and decryption; key wrapping; and key unwrapping, based on Camellia and electronic codebook mode.

14519 It does not have a parameter.

14520 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
14521 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
14522 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
14523 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
14524 length as the padded input data. It does not wrap the key type, key length, or any other information about
14525 the key; the application must convey these separately.

14526 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
14527 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
14528 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
14529 attribute of the new key; other attributes required by the key type must be specified in the template.

14530 Constraints on key types and the length of data are summarized in the following table:

14531 Table 213, 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	

14532 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14533 specify the supported range of Camellia key sizes, in bytes.

14534 **6.47.5 Camellia-CBC**

14535 Camellia-CBC, denoted **CKM_CAMELLIA_CBC**, is a mechanism for single- and multiple-part encryption
14536 and decryption; key wrapping; and key unwrapping, based on Camellia and cipher-block chaining mode.
14537 It has a parameter, a 16-byte initialization vector.

14538 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
14539 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
14540 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
14541 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
14542 length as the padded input data. It does not wrap the key type, key length, or any other information about
14543 the key; the application must convey these separately.

14544 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
14545 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
14546 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
14547 attribute of the new key; other attributes required by the key type must be specified in the template.

14548 Constraints on key types and the length of data are summarized in the following table:

14549 Table 214, 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	

14550 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14551 specify the supported range of Camellia key sizes, in bytes.

14552 **6.47.6 Camellia-CBC with PKCS padding**

14553 Camellia-CBC with PKCS padding, denoted **CKM_CAMELLIA_CBC_PAD**, is a mechanism for single-
14554 and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Camellia;
14555 cipher-block chaining mode; and the block cipher padding method detailed in [PKCS #7].

14556 It has a parameter, a 16-byte initialization vector.

14557 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the

14558 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for

14559 the **CKA_VALUE_LEN** attribute.

14560 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,

14561 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see section 6.7 for

14562 details). The entries in the table below for data length constraints when wrapping and unwrapping keys do

14563 not apply to wrapping and unwrapping private keys.

14564 Constraints on key types and the length of data are summarized in the following table:

14565 *Table 215, 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

14566 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure

14567 specify the supported range of Camellia key sizes, in bytes.

14568

14569 **6.47.7 CAMELLIA with Counter mechanism parameters**

14570 ♦ **CK_CAMELLIA_CTR_PARAMS;**

14571 **CK_CAMELLIA_CTR_PARAMS_PTR**

14572 **CK_CAMELLIA_CTR_PARAMS** is a structure that provides the parameters to the

14573 **CKM_CAMELLIA_CTR** mechanism. It is defined as follows:

```
14574     typedef struct CK_CAMELLIA_CTR_PARAMS {
14575         CK_ULONG ulCounterBits;
14576         CK_BYTE cb[16];
14577     } CK_CAMELLIA_CTR_PARAMS;
```

14579 *ulCounterBits* specifies the number of bits in the counter block (*cb*) that shall be incremented. This

14580 number shall be such that $0 < ulCounterBits \leq 128$. For any values outside this range the mechanism

14581 shall return **CKR_MECHANISM_PARAM_INVALID**.

14582 It's up to the caller to initialize all of the bits in the counter block including the counter bits. The counter

14583 bits are the least significant bits of the counter block (*cb*). They are a big-endian value usually starting

14584 with 1. The rest of 'cb' is for the nonce, and maybe an optional IV.

14585 E.g. as defined in [RFC 3686]:

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 218, 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 219, ARIA Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ARIA_KEY_GEN					✓			
CKM_ARIA_ECB	✓					✓		
CKM_ARIA_CBC	✓					✓		
CKM_ARIA_CBC_PAD	✓					✓		
CKM_ARIA_MAC_GENERAL		✓						

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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 220, 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}	CK_ULONG	Length in bytes of key value

Refer to Table 13 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)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.49.3 ARIA key generation

The ARIA key generation mechanism, denoted **CKM_ARIA_KEY_GEN**, is a key generation mechanism for Aria.

It does not have a parameter.

The mechanism generates ARIA 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 ARIA 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 ARIA key sizes, in bytes.

6.49.4 ARIA-ECB

ARIA-ECB, denoted **CKM_ARIA_ECB**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on Aria 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, 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 221, 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	

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

6.49.5 ARIA-CBC

ARIA-CBC, denoted **CKM_ARIA_CBC**, is a mechanism for single- and multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA and cipher-block chaining mode.

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

14709 This mechanism can wrap and unwrap any secret key. Of course, a particular token may not be able to
14710 wrap/unwrap every secret key that it supports. For wrapping, the mechanism encrypts the value of the
14711 **CKA_VALUE** attribute of the key that is wrapped, padded on the trailing end with up to block size minus
14712 one null bytes so that the resulting length is a multiple of the block size. The output data is the same
14713 length as the padded input data. It does not wrap the key type, key length, or any other information about
14714 the key; the application must convey these separately.

14715 For unwrapping, the mechanism decrypts the wrapped key, and truncates the result according to the
14716 **CKA_KEY_TYPE** attribute of the template and, if it has one, and the key type supports it, the
14717 **CKA_VALUE_LEN** attribute of the template. The mechanism contributes the result as the **CKA_VALUE**
14718 attribute of the new key; other attributes required by the key type must be specified in the template.

14719 Constraints on key types and the length of data are summarized in the following table:

14720 *Table 222, 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	

14721 For this mechanism, the ulMinKeySize and ulMaxKeySize fields of the CK_MECHANISM_INFO structure
14722 specify the supported range of Aria key sizes, in bytes.

14723 **6.49.6 ARIA-CBC with PKCS padding**

14724 ARIA-CBC with PKCS padding, denoted **CKM_ARIA_CBC_PAD**, is a mechanism for single- and
14725 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on ARIA; cipher-block
14726 chaining mode; and the block cipher padding method detailed in [PKCS #7].

14727 It has a parameter, a 16-byte initialization vector.

14728 The PKCS padding in this mechanism allows the length of the plaintext value to be recovered from the
14729 ciphertext value. Therefore, when unwrapping keys with this mechanism, no value should be specified for
14730 the **CKA_VALUE_LEN** attribute.

14731 In addition to being able to wrap and unwrap secret keys, this mechanism can wrap and unwrap RSA,
14732 Diffie-Hellman, X9.42 Diffie-Hellman, short Weierstrass EC and DSA private keys (see section 6.7 for
14733 details). The entries in the table below for data length constraints when wrapping and unwrapping keys do
14734 not apply to wrapping and unwrapping private keys.

14735 Constraints on key types and the length of data are summarized in the following table:

14736 Table 223, 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

14737 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14738 specify the supported range of ARIA key sizes, in bytes.

14739 **6.49.7 General-length ARIA-MAC**

14740 General-length ARIA -MAC, denoted **CKM_ARIA_MAC_GENERAL**, is a mechanism for single- and
14741 multiple-part signatures and verification, based on ARIA and data authentication as defined in [FIPS 113].

14742 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
14743 desired from the mechanism.

14744 The output bytes from this mechanism are taken from the start of the final ARIA cipher block produced in
14745 the MACing process.

14746 Constraints on key types and the length of data are summarized in the following table:

14747 Table 224, 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

14748 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14749 specify the supported range of ARIA key sizes, in bytes.

14750 **6.49.8 ARIA-MAC**

14751 ARIA-MAC, denoted by **CKM_ARIA_MAC**, is a special case of the general-length ARIA-MAC
14752 mechanism. ARIA-MAC always produces and verifies MACs that are half the block size in length.

14753 It does not have a parameter.

14754 Constraints on key types and the length of data are summarized in the following table:

14755 Table 225, 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)

14756 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
14757 specify the supported range of ARIA key sizes, in bytes.

14758 **6.50 Key derivation by data encryption - ARIA**

14759 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
14760 They are for use with the **C_DeriveKey** function.

14761 **6.50.1 Definitions**

14762 Mechanisms:

14763 CKM_ARIA_ECB_ENCRYPT_DATA

14764 CKM_ARIA_CBC_ENCRYPT_DATA

14765
14766 typedef struct CK_ARIA_CBC_ENCRYPT_DATA_PARAMS {
14767 CK_BYTE iv[16];
14768 CK_BYTE_PTR pData;
14769 CK_ULONG length;
14770 } CK_ARIA_CBC_ENCRYPT_DATA_PARAMS;
14771
14772 typedef CK_ARIA_CBC_ENCRYPT_DATA_PARAMS CK_PTR
14773 CK_ARIA_CBC_ENCRYPT_DATA_PARAMS_PTR;

14774 **6.50.2 Mechanism Parameters**

14775 Uses CK_ARIA_CBC_ENCRYPT_DATA_PARAMS, and CK_KEY_DERIVATION_STRING_DATA.

14776 Table 226, 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.

14778 **6.51 SEED**

14779 SEED is a symmetric block cipher developed by the South Korean Information Security Agency (KISA). It
14780 has a 128-bit key size and a 128-bit block size.

14781 Its specification has been published as Internet [RFC 4269].

14782 RFCs have been published defining the use of SEED in

14783 TLS <ftp://ftp.rfc-editor.org/in-notes/rfc4162.txt>

14784 IPsec <ftp://ftp.rfc-editor.org/in-notes/rfc4196.txt>

14785 CMS <ftp://ftp.rfc-editor.org/in-notes/rfc4010.txt>

14787 TLS cipher suites that use SEED include:

14788 CipherSuite TLS_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x96};
14789 CipherSuite TLS_DH_DSS_WITH_SEED_CBC_SHA = { 0x00, 0x97};
14790 CipherSuite TLS_DH_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x98};
14791 CipherSuite TLS_DHE_DSS_WITH_SEED_CBC_SHA = { 0x00, 0x99};
14792 CipherSuite TLS_DHE_RSA_WITH_SEED_CBC_SHA = { 0x00, 0x9A};
14793 CipherSuite TLS_DH_anon_WITH_SEED_CBC_SHA = { 0x00, 0x9B};

14795 As with any block cipher, it can be used in the ECB, CBC, OFB and CFB modes of operation, as well as
14796 in a MAC algorithm such as HMAC.

14797 Olds have been published for all these uses. A list may be seen at
14798 <http://www.alvestrand.no/objectid/1.2.410.200004.1.html>

14799
14800 Table 227, SEED Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
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							✓	

14801 6.51.1 Definitions

14802 This section defines the key type “**CKK_SEED**” for type CK_KEY_TYPE as used in the **CKA_KEY_TYPE**
14803 attribute of key objects.

14804 Mechanisms:

- 14805 CKM_SEED_KEY_GEN
- 14806 CKM_SEED_ECB
- 14807 CKM_SEED_CBC
- 14808 CKM_SEED_MAC
- 14809 CKM_SEED_MAC_GENERAL
- 14810 CKM_SEED_CBC_PAD

14811
14812 For all of these mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
14813 are always 16.

14814 6.51.2 SEED secret key objects

14815 SEED secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_SEED**) hold SEED keys. The
14816 following table defines the secret key object attributes, in addition to the common attributes defined for
14817 this object class:

14818 Table 228, SEED Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE ^{1,4,6,7}	Byte array	Key value (always 16 bytes long)

14819 ¹Refer to Table 13 for footnotes

14820 The following is a sample template for creating a SEED secret key object:

```
14821        CK_OBJECT_CLASS class = CKO_SECRET_KEY;  
14822        CK_KEY_TYPE keyType = CKK_SEED;  
14823        CK_UTF8CHAR label[] = "A SEED secret key object";
```

```

14824     CK_BYTE value[] = {...};
14825     CK_BBOOL true = CK_TRUE;
14826     CK_ATTRIBUTE template[] = {
14827         {CKA_CLASS, &class, sizeof(class)},
14828         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
14829         {CKA_TOKEN, &true, sizeof(true)},
14830         {CKA_LABEL, label, sizeof(label)-1},
14831         {CKA_ENCRYPT, &true, sizeof(true)},
14832         {CKA_VALUE, value, sizeof(value)}
14833     };

```

14834 6.51.3 SEED key generation

14835 The SEED key generation mechanism, denoted **CKM_SEED_KEY_GEN**, is a key generation mechanism
14836 for SEED.

14837 It does not have a parameter.

14838 The mechanism generates SEED keys.

14839 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
14840 key. Other attributes supported by the SEED key type (specifically, the flags indicating which functions
14841 the key supports) may be specified in the template for the key, or else are assigned default initial values.

14842 6.51.4 SEED-ECB

14843 SEED-ECB, denoted **CKM_SEED_ECB**, is a mechanism for single- and multiple-part encryption and
14844 decryption; key wrapping; and key unwrapping, based on SEED and electronic codebook mode.

14845 It does not have a parameter.

14846 6.51.5 SEED-CBC

14847 SEED-CBC, denoted **CKM_SEED_CBC**, is a mechanism for single- and multiple-part encryption and
14848 decryption; key wrapping; and key unwrapping, based on SEED and cipher-block chaining mode.

14849 It has a parameter, a 16-byte initialization vector.

14850 6.51.6 SEED-CBC with PKCS padding

14851 SEED-CBC with PKCS padding, denoted **CKM_SEED_CBC_PAD**, is a mechanism for single- and
14852 multiple-part encryption and decryption; key wrapping; and key unwrapping, based on SEED; cipher-
14853 block chaining mode; and the block cipher padding method detailed in [PKCS #7].

14854 It has a parameter, a 16-byte initialization vector.

14855 6.51.7 General-length SEED-MAC

14856 General-length SEED-MAC, denoted **CKM_SEED_MAC_GENERAL**, is a mechanism for single- and
14857 multiple-part signatures and verification, based on SEED and data authentication.

14858 It has a parameter, a **CK_MAC_GENERAL_PARAMS** structure, which specifies the output length
14859 desired from the mechanism.

14860 The output bytes from this mechanism are taken from the start of the final cipher block produced in the
14861 MACing process.

14862 6.51.8 SEED-MAC

14863 SEED-MAC, denoted by **CKM_SEED_MAC**, is a special case of the general-length SEED-MAC
14864 mechanism. SEED-MAC always produces and verifies MACs that are half the block size in length.

14865 It does not have a parameter.

14866 **6.52 Key derivation by data encryption - SEED**

14867 These mechanisms allow derivation of keys using the result of an encryption operation as the key value.
14868 They are for use with the **C_DeriveKey** function.

14869 **6.52.1 Definitions**

14870 Mechanisms:

14871 CKM_SEED_ECB_ENCRYPT_DATA

14872 CKM_SEED_CBC_ENCRYPT_DATA

14873

14874 typedef struct CK_SEED_CBC_ENCRYPT_DATA_PARAMS {
14875 CK_BYTE iv[16];
14876 CK_BYTE_PTR pData;
14877 CK_ULONG length;
14878 } CK_SEED_CBC_ENCRYPT_DATA_PARAMS;

14879

14880 typedef CK_SEED_CBC_ENCRYPT_DATA_PARAMS CK_PTR
14881 CK_SEED_CBC_ENCRYPT_DATA_PARAMS_PTR;

14882 **6.52.2 Mechanism Parameters**

14883 *Table 229, 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.

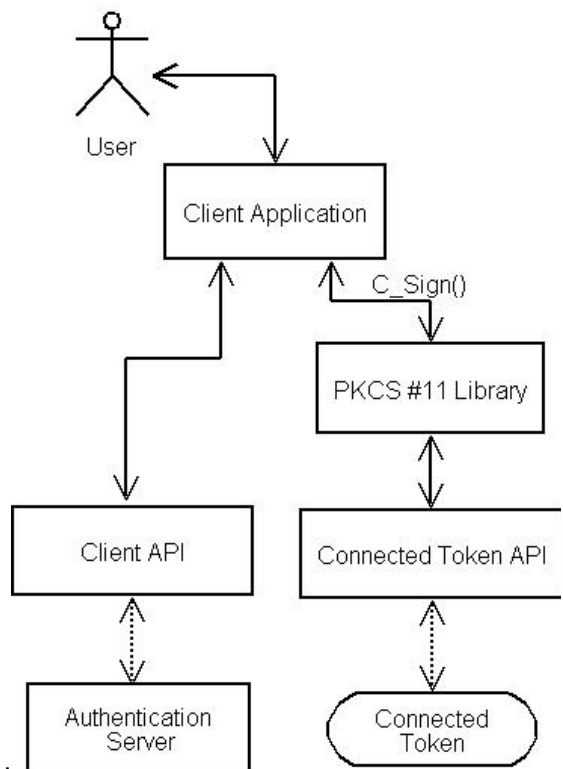
14884

14885 **6.53 OTP**

14886 **6.53.1 Usage overview**

14887 OTP tokens represented as PKCS #11 mechanisms may be used in a variety of ways. The usage cases
14888 can be categorized according to the type of sought functionality.

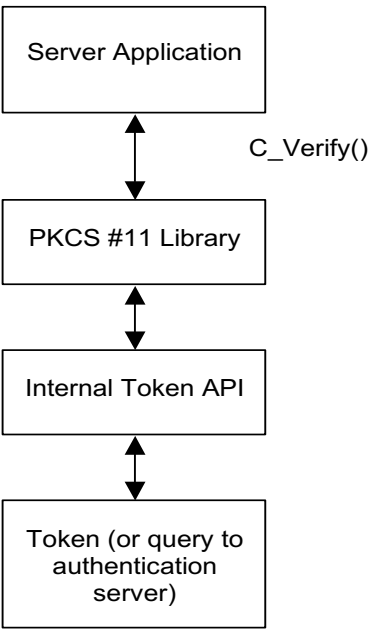
14889 **6.53.2 Case 1: Generation of OTP values**



14890
14891 *Figure 2: Retrieving OTP values through C_Sign*

14892 Figure 2 shows an integration of PKCS #11 into an application that needs to authenticate users holding
14893 OTP tokens. In this particular example, a connected hardware token is used, but a software token is
14894 equally possible. The application invokes **C_Sign** to retrieve the OTP value from the token. In the
14895 example, the application then passes the retrieved OTP value to a client API that sends it via the network
14896 to an authentication server. The client API may implement a standard authentication protocol such as
14897 RADIUS [RFC 2865] or EAP [RFC 3748], or a proprietary protocol such as that used by RSA Security's
14898 ACE/Agent® software.

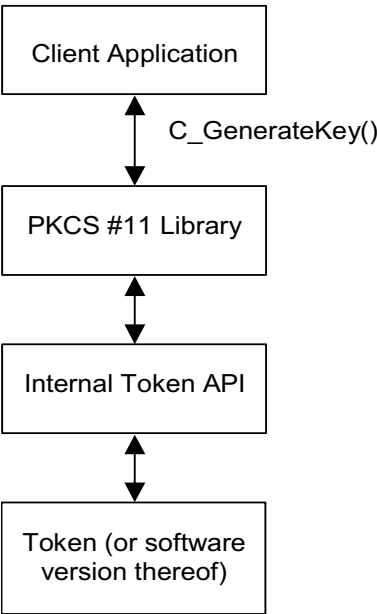
14899 **6.53.3 Case 2: Verification of provided OTP values**



14900
14901 *Figure 3: Server-side verification of OTP values*

14902 Figure 3 illustrates the server-side equivalent of the scenario depicted in Figure 2. In this case, a server
14903 application invokes **C_Verify** with the received OTP value as the signature value to be verified.

14904 **6.53.4 Case 3: Generation of OTP keys**



14905
14906 *Figure 4: Generation of an OTP key*

14907 Figure 4 shows an integration of PKCS #11 into an application that generates OTP keys. The application
14908 invokes **C_GenerateKey** to generate an OTP key of a particular type on the token. The key may
14909 subsequently be used as a basis to generate OTP values.

14910 **6.53.5 OTP objects**

14911 **6.53.5.1 Key objects**

14912 OTP key objects (object class **CKO_OTP_KEY**) hold secret keys used by OTP tokens. The following
14913 table defines the attributes common to all OTP keys, in addition to the attributes defined for secret keys,
14914 all of which are inherited by this class:

14915 *Table 230, 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:

		<p>CK_OTP_PARAM_MANDATORY = A PIN value must be supplied.</p> <p>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)</p> <p>CK_OTP_PARAM_IGNORED = A PIN value, if supplied, will be ignored.</p>
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.

14916 Refer to Table 13 for footnotes

14917 Note: A Cryptoki library may support PIN-code caching in order to reduce user interactions. An OTP-
14918 PKCS #11 application should therefore always consult the state of the **CKA_OTP_PIN_REQUIREMENT**
14919 attribute before each call to **C_SignInit**, as the value of this attribute may change dynamically.

14920 For OTP tokens with multiple keys, the keys may be enumerated using **C_FindObjects**. The
14921 **CKA_OTP_SERVICE_IDENTIFIER** and/or the **CKA_OTP_SERVICE_LOGO** attribute may be used to
14922 distinguish between keys. The actual choice of key for a particular operation is however application-
14923 specific and beyond the scope of this document.

14924 For all OTP keys, the **CKA_ALLOWED_MECHANISMS** attribute should be set as required.

14925 6.53.6 OTP-related notifications

14926 This document extends the set of defined notifications as follows:

14927	CKN_OTP_CHANGED	Cryptoki is informing the application that the OTP for a key on a
14928		connected token just changed. This notification is particularly useful
14929		when applications wish to display the current OTP value for time-
14930		based mechanisms.

14931 6.53.7 OTP mechanisms

14932 The following table shows, for the OTP mechanisms defined in this document, their support by different
14933 cryptographic operations. For any particular token, of course, a particular operation may well support only
14934 a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism

14935 for some operation supports any other mechanism for any other operation (or even supports that same
14936 mechanism for any other operation).
14937 *Table 231, OTP mechanisms vs. applicable functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SECURID_KEY_GEN					✓			
CKM_SECURID		✓						
CKM_HOTP_KEY_GEN					✓			
CKM_HOTP		✓						
CKM_ACTI_KEY_GEN					✓			
CKM_ACTI		✓						

14938 The remainder of this section will present in detail the OTP mechanisms and the parameters that are
14939 supplied to them.

14940 **6.53.7.1 OTP mechanism parameters**

14941 **◆ CK_OTP_PARAM_TYPE**

14942 **CK_OTP_PARAM_TYPE** is a value that identifies an OTP parameter type. It is defined as follows:

14943 `typedef CK_ULONG CK_OTP_PARAM_TYPE;`

14944 The following **CK_OTP_PARAM_TYPE** types are defined:

14945 *Table 232, 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

Parameter	Data type	Meaning
		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.

14946

14947 The following table defines the possible values for the CK_OTP_FLAGS type:

14948 *Table 233, 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 .
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.

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_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:

```
typedef struct CK_OTP_PARAM {
    CK_OTP_PARAM_TYPE type;
    CK_VOID_PTR pValue;
    CK_ULONG ulValueLen;
} CK_OTP_PARAM;
```

The fields of the structure have the following meanings:

type	the parameter type
pValue	pointer to the value of the parameter
ulValueLen	length in bytes of the value

If a parameter has no value, then *ulValueLen* = 0, and the value of *pValue* is irrelevant. Note that *pValue* is a “void” pointer, facilitating the passing of arbitrary values. Both the application and the Cryptoki library must ensure that the pointer can be safely cast to the expected type (i.e., without word-alignment errors).

CK_OTP_PARAM_PTR is a pointer to a **CK_OTP_PARAM**.

◆ CK_OTP_PARAMS; CK_OTP_PARAMS_PTR

CK_OTP_PARAMS is a structure that is used to provide parameters for OTP mechanisms in a generic fashion. It is defined as follows:

```
typedef struct CK_OTP_PARAMS {  
    CK_OTP_PARAM_PTR pParams;  
    CK_ULONG ulCount;  
} CK_OTP_PARAMS;
```

The fields of the structure have the following meanings:

pParams	pointer to an array of OTP parameters
ulCount	the number of parameters in the array

CK_OTP_PARAMS_PTR is a pointer to a **CK_OTP_PARAMS**.

When calling **C_SignInit** or **C_VerifyInit** with a mechanism that takes a **CK_OTP_PARAMS** structure as a parameter, the **CK_OTP_PARAMS** structure shall be populated in accordance with the **CKA_OTP_X_REQUIREMENT** key attributes for the identified key, where **X** is PIN, CHALLENGE, TIME, or COUNTER.

For example, if **CKA_OTP_TIME_REQUIREMENT** = **CK_OTP_PARAM_MANDATORY**, then the **CK_OTP_TIME** parameter shall be present. If **CKA_OTP_TIME_REQUIREMENT** = **CK_OTP_PARAM_OPTIONAL**, then a **CK_OTP_TIME** parameter may be present. If it is not present, then the library may collect it (during the **C_Sign** call). If **CKA_OTP_TIME_REQUIREMENT** = **CK_OTP_PARAM_IGNORED**, then a provided **CK_OTP_TIME** parameter will always be ignored. Additionally, a provided **CK_OTP_TIME** parameter will always be ignored if **CKF_EXCLUDE_TIME** is set in a **CK_OTP_FLAGS** parameter. Similarly, if this flag is set, a library will not attempt to collect the value itself, and it will also instruct the token not to make use of any internal value, subject to token policies. It is an error (**CKR_MECHANISM_PARAM_INVALID**) to set the **CKF_EXCLUDE_TIME** flag when the **CKA_OTP_TIME_REQUIREMENT** attribute is **CK_OTP_PARAM_MANDATORY**.

The above discussion holds for all **CKA_OTP_X_REQUIREMENT** attributes (*i.e.*, **CKA_OTP_PIN_REQUIREMENT**, **CKA_OTP_CHALLENGE_REQUIREMENT**, **CKA_OTP_COUNTER_REQUIREMENT**, **CKA_OTP_TIME_REQUIREMENT**). A library may set a particular **CKA_OTP_X_REQUIREMENT** attribute to **CK_OTP_PARAM_OPTIONAL** even if it is required by the mechanism as long as the token (or the library itself) has the capability of providing the value to the 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 234, 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.

Refer to Table 13 for footnotes

The following is a sample template for creating an RSA SecurID secret key object:

```
CK_OBJECT_CLASS class = CKO_OTP_KEY;
CK_KEY_TYPE keyType = CKK_SECURID;
CK_DATE endDate = {...};
CK_UTF8CHAR label[] = "RSA SecurID secret key object";
CK_BYTE keyId[] = {...};
CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
CK_ULONG outputLength = 6;
CK_ULONG needPIN = CK_OTP_PARAM_MANDATORY;
CK_ULONG timeInterval = 60;
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)},
```

```

15062         {CKA_SENSITIVE, &true, sizeof(true)},
15063         {CKA_LABEL, label, sizeof(label)-1},
15064         {CKA_SIGN, &true, sizeof(true)},
15065         {CKA_VERIFY, &true, sizeof(true)},
15066         {CKA_ID, keyId, sizeof(keyId)},
15067         {CKA_OTP_FORMAT, &outputFormat, sizeof(outputFormat)},
15068         {CKA_OTP_LENGTH, &outputLength, sizeof(outputLength)},
15069         {CKA_OTP_PIN_REQUIREMENT, &needPIN, sizeof(needPIN)},
15070         {CKA_OTP_TIME_INTERVAL, &timeInterval,
15071          sizeof(timeInterval)},
15072         {CKA_VALUE, value, sizeof(value)}
15073     };

```

15074 6.53.8.2 RSA SecurID key generation

15075 The RSA SecurID key generation mechanism, denoted **CKM_SECURID_KEY_GEN**, is a key generation
15076 mechanism for the RSA SecurID algorithm.

15077 It does not have a parameter.

15078 The mechanism generates RSA SecurID keys with a particular set of attributes as specified in the
15079 template for the key.

15080 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE_LEN**, and
15081 **CKA_VALUE** attributes to the new key. Other attributes supported by the RSA SecurID key type may be
15082 specified in the template for the key, or else are assigned default initial values.

15083 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15084 specify the supported range of SecurID key sizes, in bytes.

15085 6.53.8.3 SecurID OTP generation and validation

15086 **CKM_SECURID** is the mechanism for the retrieval and verification of RSA SecurID OTP values.

15087 The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

15088 When signing or verifying using the **CKM_SECURID** mechanism, *pData* shall be set to **NULL_PTR** and
15089 *ulDataLen* shall be set to 0.

15090 6.53.8.4 Return values

15091 Support for the **CKM_SECURID** mechanism extends the set of return values for **C_Verify** with the
15092 following values:

- 15093 • **CKR_NEW_PIN_MODE**: The supplied OTP was not accepted, and the library requests a new OTP
15094 computed using a new PIN. The new PIN is set through means out of scope for this document.
- 15095 • **CKR_NEXT_OTP**: The supplied OTP was correct but indicated a larger than normal drift in the
15096 token's internal state (e.g. clock, counter). To ensure this was not due to a temporary problem, the
15097 application should provide the next one-time password to the library for verification.

15098 6.53.9 OATH HOTP

15099 6.53.9.1 OATH HOTP secret key objects

15100 HOTP secret key objects (object class **CKO_OTP_KEY**, key type **CKK_HOTP**) hold generic secret keys
15101 and associated counter values.

15102 The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a
15103 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.

15150 As for the **CKM_SECURID** mechanism, when signing or verifying using the **CKM_HOTP** mechanism,
 15151 *pData* shall be set to **NULL_PTR** and *ulDataLen* shall be set to 0.

15152 For verify operations, the counter value **CK_OTP_COUNTER** must be provided as a **CK_OTP_PARAM**
 15153 parameter to **C_VerifyInit**. When verifying an OTP value using the **CKM_HOTP** mechanism, *pSignature*
 15154 shall be set to the OTP value itself, e.g. the value of the **CK_OTP_VALUE** component of a
 15155 **CK_OTP_PARAM** structure in the case of an earlier call to **C_Sign**.

15156 6.53.10 ActivIdentity ACTI

15157 6.53.10.1 ACTI secret key objects

15158 ACTI secret key objects (object class **CKO_OTP_KEY**, key type **CKK_ACTI**) hold ActivIdentity ACTI
 15159 secret keys.

15160 For ACTI keys, the **CKA_OTP_COUNTER** value shall be an 8 bytes unsigned integer in big endian (i.e.
 15161 network byte order) form. The same holds true for the **CK_OTP_COUNTER** value in the
 15162 **CK_OTP_PARAM** structure.

15163 The **CKA_OTP_COUNTER** value may be set at key generation; however, some tokens may set it to a
 15164 fixed initial value. Depending on the token's security policy, this value may not be modified and/or may
 15165 not be revealed if the object has its **CKA_SENSITIVE** attribute set to **CK_TRUE** or its
 15166 **CKA_EXTRACTABLE** attribute set to **CK_FALSE**.

15167 The **CKA_OTP_TIME** value may be set at key generation; however, some tokens may set it to a fixed
 15168 initial value. Depending on the token's security policy, this value may not be modified and/or may not be
 15169 revealed if the object has its **CKA_SENSITIVE** attribute set to **CK_TRUE** or its **CKA_EXTRACTABLE**
 15170 attribute set to **CK_FALSE**.

15171 The following is a sample template for creating an ACTI secret key object:

```

15172     CK_OBJECT_CLASS class = CKO_OTP_KEY;
15173     CK_KEY_TYPE keyType = CKK_ACTI;
15174     CK_UTF8CHAR label[] = "ACTI secret key object";
15175     CK_BYTE keyId[] = {...};
15176     CK_ULONG outputFormat = CK_OTP_FORMAT_DECIMAL;
15177     CK_ULONG outputLength = 6;
15178     CK_DATE endDate = {...};
15179     CK_BYTE counterValue[8] = {0};
15180     CK_BYTE value[] = {...};
15181     CK_BBOOL true = CK_TRUE;
15182     CK_ATTRIBUTE template[] = {
15183         {CKA_CLASS, &class, sizeof(class)},
15184         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
15185         {CKA_END_DATE, &endDate, sizeof(endDate)},
15186         {CKA_TOKEN, &true, sizeof(true)},
15187         {CKA_SENSITIVE, &true, sizeof(true)},
15188         {CKA_LABEL, label, sizeof(label)-1},
15189         {CKA_SIGN, &true, sizeof(true)},
15190         {CKA_VERIFY, &true, sizeof(true)},
15191         {CKA_ID, keyId, sizeof(keyId)},
15192         {CKA_OTP_FORMAT, &outputFormat,
15193          sizeof(outputFormat)},
15194         {CKA_OTP_LENGTH, &outputLength,
15195          sizeof(outputLength)},
15196         {CKA_OTP_COUNTER, counterValue,

```

```
15197         sizeof(counterValue) },  
15198         {CKA_VALUE, value, sizeof(value)}  
15199     };
```

15200 6.53.10.2 ACTI key generation

15201 The ACTI key generation mechanism, denoted **CKM_ACTI_KEY_GEN**, is a key generation mechanism
15202 for the ACTI algorithm.

15203 It does not have a parameter.

15204 The mechanism generates ACTI keys with a particular set of attributes as specified in the template for the
15205 key.

15206 The mechanism contributes at least the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE** and
15207 **CKA_VALUE_LEN** attributes to the new key. Other attributes supported by the ACTI key type may be
15208 specified in the template for the key, or else are assigned default initial values.

15209 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15210 specify the supported range of ACTI key sizes, in bytes.

15211 6.53.10.3 ACTI OTP generation and validation

15212 **CKM_ACTI** is the mechanism for the retrieval and verification of ACTI OTP values.

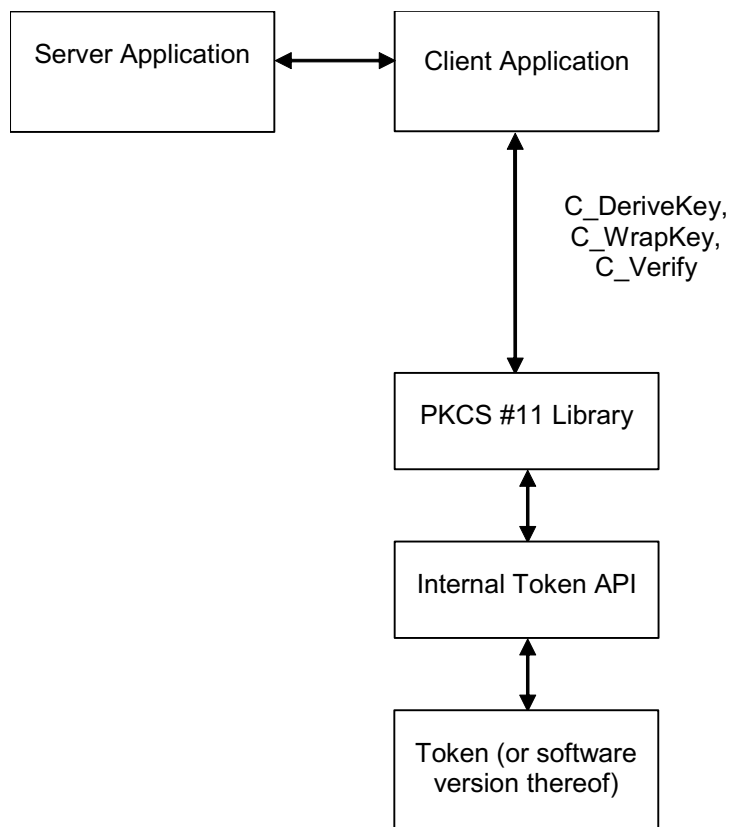
15213 The mechanism takes a pointer to a **CK_OTP_PARAMS** structure as a parameter.

15214 When signing or verifying using the **CKM_ACTI** mechanism, *pData* shall be set to **NULL_PTR** and
15215 *ulDataLen* shall be set to 0.

15216 When verifying an OTP value using the **CKM_ACTI** mechanism, *pSignature* shall be set to the OTP value
15217 itself, e.g. the value of the **CK_OTP_VALUE** component of a **CK_OTP_PARAM** structure in the case of
15218 an earlier call to **C_Sign**.

15219 **6.54 CT-KIP**

15220 **6.54.1 Principles of Operation**



15221
15222 *Figure 5: PKCS #11 and CT-KIP integration*

15223 Figure 5 shows an integration of PKCS #11 into an application that generates cryptographic keys through
15224 the use of CT-KIP. The application invokes **C_DeriveKey** to derive a key of a particular type on the token.
15225 The key may subsequently be used as a basis to e.g., generate one-time password values. The
15226 application communicates with a CT-KIP server that participates in the key derivation and stores a copy
15227 of the key in its database. The key is transferred to the server in wrapped form, after a call to
15228 **C_WrapKey**. The server authenticates itself to the client and the client verifies the authentication by calls
15229 to **C_Verify**.

15230 **6.54.2 Mechanisms**

15231 The following table shows, for the mechanisms defined in this document, their support by different
15232 cryptographic operations. For any particular token, of course, a particular operation may well support only
15233 a subset of the mechanisms listed. There is also no guarantee that a token that supports one mechanism
15234 for some operation supports any other mechanism for any other operation (or even supports that same
15235 mechanism for any other operation).

15236 Table 235, CT-KIP Mechanisms vs. applicable functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_KIP_DERIVE							✓	
CKM_KIP_WRAP						✓		
CKM_KIP_MAC		✓						

15237 The remainder of this section will present in detail the mechanisms and the parameters that are supplied
15238 to them.

15239 6.54.3 Definitions

15240 Mechanisms:

- 15241 CKM_KIP_DERIVE
- 15242 CKM_KIP_WRAP
- 15243 CKM_KIP_MAC

15244 6.54.4 CT-KIP Mechanism parameters

15245 ♦ CK_KIP_PARAMS; CK_KIP_PARAMS_PTR

15246 **CK_KIP_PARAMS** is a structure that provides the parameters to all the CT-KIP related mechanisms: The
15247 **CKM_KIP_DERIVE** key derivation mechanism, the **CKM_KIP_WRAP** key wrap and key unwrap
15248 mechanism, and the **CKM_KIP_MAC** signature mechanism. The structure is defined as follows:

```
15249 typedef struct CK_KIP_PARAMS {  
15250     CK_MECHANISM_PTR    pMechanism;  
15251     CK_OBJECT_HANDLE    hKey;  
15252     CK_BYTE_PTR         pSeed;  
15253     CK_ULONG            ulSeedLen;  
15254 } CK_KIP_PARAMS;
```

15255 The fields of the structure have the following meanings:

- 15256 pMechanism pointer to the underlying cryptographic mechanism (e.g. AES, SHA-
15257 256)
- 15258 hKey handle to a key that will contribute to the entropy of the derived key
15259 (CKM_KIP_DERIVE) or will be used in the MAC operation
15260 (CKM_KIP_MAC)
- 15261 pSeed pointer to an input seed
- 15262 ulSeedLen length in bytes of the input seed

15263 **CK_KIP_PARAMS_PTR** is a pointer to a **CK_KIP_PARAMS** structure.

15264 6.54.5 CT-KIP key derivation

15265 The CT-KIP key derivation mechanism, denoted **CKM_KIP_DERIVE**, is a key derivation mechanism that
15266 is capable of generating secret keys of potentially any type, subject to token limitations.

15267 It takes a parameter of type **CK_KIP_PARAMS** which allows for the passing of the desired underlying
15268 cryptographic mechanism as well as some other data. In particular, when the *hKey* parameter is a handle

15269 to an existing key, that key will be used in the key derivation in addition to the *hBaseKey* of **C_DeriveKey**.
15270 The *pSeed* parameter may be used to seed the key derivation operation.
15271 The mechanism derives a secret key with a particular set of attributes as specified in the attributes of the
15272 template for the key.
15273 The mechanism contributes the **CKA_CLASS** and **CKA_VALUE** attributes to the new key. Other
15274 attributes supported by the key type may be specified in the template for the key, or else will be assigned
15275 default initial values. Since the mechanism is generic, the **CKA_KEY_TYPE** attribute should be set in the
15276 template, if the key is to be used with a particular mechanism.

15277 **6.54.6 CT-KIP key wrap and key unwrap**

15278 The CT-KIP key wrap and unwrap mechanism, denoted **CKM_KIP_WRAP**, is a key wrap mechanism that
15279 is capable of wrapping and unwrapping generic secret keys.
15280 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
15281 cryptographic mechanism as well as some other data. It does not make use of the *hKey* parameter of
15282 **CK_KIP_PARAMS**.

15283 **6.54.7 CT-KIP signature generation**

15284 The CT-KIP signature (MAC) mechanism, denoted **CKM_KIP_MAC**, is a mechanism used to produce a
15285 message authentication code of arbitrary length. The keys it uses are secret keys.
15286 It takes a parameter of type **CK_KIP_PARAMS**, which allows for the passing of the desired underlying
15287 cryptographic mechanism as well as some other data. The mechanism does not make use of the *pSeed*
15288 and the *ulSeedLen* parameters of **CT_KIP_PARAMS**.
15289 This mechanism produces a MAC of the length specified by *puSignatureLen* parameter in calls to
15290 **C_Sign**.
15291 If a call to **C_Sign** with this mechanism fails, then no output will be generated.

15292 **6.55 GOST 28147-89**

15293 GOST 28147-89 is a block cipher with 64-bit block size and 256-bit keys.

15294

15295 *Table 236, GOST 28147-89 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_GOST28147_KEY_GEN					✓			
CKM_GOST28147_ECB	✓					✓		
CKM_GOST28147	✓					✓		
CKM_GOST28147_MAC		✓						
CKM_GOST28147_KEY_WRAP						✓		

15296

15297 **6.55.1 Definitions**

15298 This section defines the key type “**CKK_GOST28147**” for type **CK_KEY_TYPE** as used in the
15299 **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

- 15300 Mechanisms:
- 15301 CKM_GOST28147_KEY_GEN

15302 CKM_GOST28147_ECB

15303 CKM_GOST28147

15304 CKM_GOST28147_MAC

15305 CKM_GOST28147_KEY_WRAP

15306

6.55.2 GOST 28147-89 secret key objects

15307 GOST 28147-89 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_GOST28147**) hold

15308 GOST 28147-89 keys. The following table defines the GOST 28147-89 secret key object attributes, in

15309 addition to the common attributes defined for this object class:

15310

Table 237, GOST 28147-89 Secret Key Object Attributes

Attribute	Data type	Meaning
CKA_VALUE1,4,6,7	Byte array	32 bytes in little endian order
CKA_GOST28147_PARAMS1,3,5	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

- 15311 Refer to Table 13 for footnotes
- 15312 The following is a sample template for creating a GOST 28147-89 secret key object:
- 15313 CK_OBJECT_CLASS class = CKO_SECRET_KEY;

15314 CK_KEY_TYPE keyType = CKK_GOST28147;

15315 CK_UTF8CHAR label[] = "A GOST 28147-89 secret key object";

15316 CK_BYTE value[32] = {...};

15317 CK_BYTE params_oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02,

15318 0x02, 0x1f, 0x00};

15319 CK_BBOOL true = CK_TRUE;

15320 CK_ATTRIBUTE template[] = {

15321 {CKA_CLASS, &class, sizeof(class)},

15322 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},

15323 {CKA_TOKEN, &>true, sizeof(true)},

15324 {CKA_LABEL, label, sizeof(label)-1},

15325 {CKA_ENCRYPT, &>true, sizeof(true)},

15326 {CKA_GOST28147_PARAMS, params_oid, sizeof(params_oid)},

15327 {CKA_VALUE, value, sizeof(value)}

15328 };

15329

6.55.3 GOST 28147-89 domain parameter objects

15330 GOST 28147-89 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type

15331 **CKK_GOST28147**) hold GOST 28147-89 domain parameters.

15332 The following table defines the GOST 28147-89 domain parameter object attributes, in addition to the

15333 common attributes defined for this object class:

15334 Table 238, 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 Gost28147-89-ParamSetParameters)
CKA_OBJECT_ID ¹	Byte array	DER-encoding of the object identifier indicating the domain parameters

15335 ¹Refer to Table 13 for footnotes

15336 For any particular token, there is no guarantee that a token supports domain parameters loading up
15337 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
15338 take in account that **CKA_VALUE** attribute may be inaccessible.

15339 The following is a sample template for creating a GOST 28147-89 domain parameter object:

```
15340 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;  
15341 CK_KEY_TYPE keyType = CKK_GOST28147;  
15342 CK_UTF8CHAR label[] = "A GOST 28147-89 cryptographic  
15343     parameters object";  
15344 CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,  
15345     0x1f, 0x00};  
15346 CK_BYTE value[] = {  
15347     0x30, 0x62, 0x04, 0x40, 0x4c, 0xde, 0x38, 0x9c, 0x29, 0x89, 0xef, 0xb6,  
15348     0xff, 0xeb, 0x56, 0xc5, 0x5e, 0xc2, 0x9b, 0x02, 0x98, 0x75, 0x61, 0x3b,  
15349     0x11, 0x3f, 0x89, 0x60, 0x03, 0x97, 0x0c, 0x79, 0x8a, 0xa1, 0xd5, 0x5d,  
15350     0xe2, 0x10, 0xad, 0x43, 0x37, 0x5d, 0xb3, 0x8e, 0xb4, 0x2c, 0x77, 0xe7,  
15351     0xcd, 0x46, 0xca, 0xfa, 0xd6, 0x6a, 0x20, 0x1f, 0x70, 0xf4, 0x1e, 0xa4,  
15352     0xab, 0x03, 0xf2, 0x21, 0x65, 0xb8, 0x44, 0xd8, 0x02, 0x01, 0x00, 0x02,  
15353     0x01, 0x40, 0x30, 0x0b, 0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x0e,  
15354     0x00, 0x05, 0x00  
15355 };  
15356 CK_BBOOL true = CK_TRUE;  
15357 CK_ATTRIBUTE template[] = {  
15358     {CKA_CLASS, &class, sizeof(class)},  
15359     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
15360     {CKA_TOKEN, &true, sizeof(true)},  
15361     {CKA_LABEL, label, sizeof(label)-1},  
15362     {CKA_OBJECT_ID, oid, sizeof(oid)},  
15363     {CKA_VALUE, value, sizeof(value)}  
15364 };
```

15365 **6.55.4 GOST 28147-89 key generation**

15366 The GOST 28147-89 key generation mechanism, denoted **CKM_GOST28147_KEY_GEN**, is a key
15367 generation mechanism for GOST 28147-89.

15368 It does not have a parameter.

15369 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
15370 key. Other attributes supported by the GOST 28147-89 key type may be specified for objects of object
15371 class **CKO_SECRET_KEY**.

15372 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** are not
15373 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 239, 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 240, 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 241, 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:

15440 Table 242, 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

15441
15442 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15443 are not used.
15444

15445 **6.56 GOST R 34.11-94**

15446 GOST R 34.11-94 is a mechanism for message digesting, following the hash algorithm with 256-bit
15447 message digest defined in [GOST R 34.11-94].

15448
15449 Table 243, GOST R 34.11-94 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_GOSTR3411				✓				
CKM_GOSTR3411_HMAC		✓						

15450
15451 **6.56.1 Definitions**

15452 This section defines the key type “**CKK_GOSTR3411**” for type CK_KEY_TYPE as used in the
15453 **CKA_KEY_TYPE** attribute of domain parameter objects.
15454 Mechanisms:
15455 CKM_GOSTR3411
15456 CKM_GOSTR3411_HMAC

15457 **6.56.2 GOST R 34.11-94 domain parameter objects**

15458 GOST R 34.11-94 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
15459 **CKK_GOSTR3411**) hold GOST R 34.11-94 domain parameters.
15460 The following table defines the GOST R 34.11-94 domain parameter object attributes, in addition to the
15461 common attributes defined for this object class:

15462 Table 244, 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

15463 ¹Refer to Table 13 for footnotes

15464 For any particular token, there is no guarantee that a token supports domain parameters loading up
15465 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
15466 take in account that **CKA_VALUE** attribute may be inaccessible.

15467 The following is a sample template for creating a GOST R 34.11-94 domain parameter object:

```
15468 CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;  
15469 CK_KEY_TYPE keyType = CKK_GOSTR3411;  
15470 CK_UTF8CHAR label[] = "A GOST R34.11-94 cryptographic  
15471 parameters object";  
15472 CK_BYTE oid[] = {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02,  
15473 0x1e, 0x00};  
15474 CK_BYTE value[] = {  
15475 0x30, 0x64, 0x04, 0x40, 0x4e, 0x57, 0x64, 0xd1, 0xab, 0x8d, 0xcb, 0xbf,  
15476 0x94, 0x1a, 0x7a, 0x4d, 0x2c, 0xd1, 0x10, 0x10, 0xd6, 0xa0, 0x57, 0x35,  
15477 0x8d, 0x38, 0xf2, 0xf7, 0x0f, 0x49, 0xd1, 0x5a, 0xea, 0x2f, 0x8d, 0x94,  
15478 0x62, 0xee, 0x43, 0x09, 0xb3, 0xf4, 0xa6, 0xa2, 0x18, 0xc6, 0x98, 0xe3,  
15479 0xc1, 0x7c, 0xe5, 0x7e, 0x70, 0x6b, 0x09, 0x66, 0xf7, 0x02, 0x3c, 0x8b,  
15480 0x55, 0x95, 0xbf, 0x28, 0x39, 0xb3, 0x2e, 0xcc, 0x04, 0x20, 0x00, 0x00,  
15481 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
15482 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
15483 0x00, 0x00, 0x00, 0x00, 0x00, 0x00  
15484 };  
15485 CK_BBOOL true = CK_TRUE;  
15486 CK_ATTRIBUTE template[] = {  
15487 {CKA_CLASS, &class, sizeof(class)},  
15488 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
15489 {CKA_TOKEN, &true, sizeof(true)},  
15490 {CKA_LABEL, label, sizeof(label)-1},  
15491 {CKA_OBJECT_ID, oid, sizeof(oid)},  
15492 {CKA_VALUE, value, sizeof(value)}  
15493 };
```

15494 **6.56.3 GOST R 34.11-94 digest**

15495 GOST R 34.11-94 digest, denoted **CKM_GOSTR3411**, is a mechanism for message digesting based on
15496 GOST R 34.11-94 hash algorithm [GOST R 34.11-94].

15497 As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter
15498 may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357]
15499 (section 11.2) must be used.

15500 Constraints on the length of input and output data are summarized in the following table. For single-part
15501 digesting, the data and the digest may begin at the same location in memory.

15502 Table 245, GOST R 34.11-94: Data Length

Function	Input length	Digest length
C_Digest	Any	32 bytes

15503
15504 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15505 are not used.

15506 **6.56.4 GOST R 34.11-94 HMAC**

15507 GOST R 34.11-94 HMAC mechanism, denoted **CKM_GOSTR3411_HMAC**, is a mechanism for
15508 signatures and verification. It uses the HMAC construction, based on the GOST R 34.11-94 hash function
15509 [GOST R 34.11-94] and core HMAC algorithm [RFC 2104]. The keys it uses are of generic key type
15510 **CKK_GENERIC_SECRET** or **CKK_GOST28147**.

15511 To be conformed to GOST R 34.11-94 hash algorithm [GOST R 34.11-94] the block length of core HMAC
15512 algorithm is 32 bytes long (see [RFC 2104] section 2, and [RFC 4357] section 3).

15513 As a parameter this mechanism utilizes a DER-encoding of the object identifier. A mechanism parameter
15514 may be missed then parameters of the object identifier *id-GostR3411-94-CryptoProParamSet* [RFC 4357]
15515 (section 11.2) must be used.

15516 Signatures (MACs) produced by this mechanism are of 32 bytes long.

15517 Constraints on the length of input and output data are summarized in the following table:

15518 Table 246, 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

15519 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15520 are not used.

15521 **6.57 GOST R 34.10-2001**

15522 GOST R 34.10-2001 is a mechanism for single- and multiple-part signatures and verification, following
15523 the digital signature algorithm defined in [GOST R 34.10-2001].

15524
15525 Table 247, GOST R34.10-2001 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_GOSTR3410_KEY_PAIR_GEN					✓			
CKM_GOSTR3410		✓ ¹						
CKM_GOSTR3410_WITH_GOSTR3411		✓						
CKM_GOSTR3410_KEY_WRAP						✓		
CKM_GOSTR3410_DERIVE							✓	

15526 1 Single-part operations only
15527

15528 **6.57.1 Definitions**

15529 This section defines the key type “**CKK_GOSTR3410**” for type CK_KEY_TYPE as used in the
15530 **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.

15531 Mechanisms:

- 15532 CKM_GOSTR3410_KEY_PAIR_GEN
- 15533 CKM_GOSTR3410
- 15534 CKM_GOSTR3410_WITH_GOSTR3411
- 15535 CKM_GOSTR3410
- 15536 CKM_GOSTR3410_KEY_WRAP
- 15537 CKM_GOSTR3410_DERIVE

15538 **6.57.2 GOST R 34.10-2001 public key objects**

15539 GOST R 34.10-2001 public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_GOSTR3410**)
15540 hold GOST R 34.10-2001 public keys.

15541 The following table defines the GOST R 34.10-2001 public key object attributes, in addition to the
15542 common attributes defined for this object class:

15543 *Table 248, 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

15544 ¹ Refer to Table 13 for footnotes

15545 The following is a sample template for creating an GOST R 34.10-2001 public key object:

15546 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;

15547 CK_KEY_TYPE keyType = CKK_GOSTR3410;

15548 CK_UTF8CHAR label[] = "A GOST R34.10-2001 public key object";

15549 CK_BYTE gostR3410params_oid[] =

15550 {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};

15551 CK_BYTE gostR3411params_oid[] =

15552 {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};

15553 CK_BYTE gost28147params_oid[] =

15554 {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};

15555 CK_BYTE value[64] = {...};

15556 CK_BBOOL true = CK_TRUE;

15557 CK_ATTRIBUTE template[] = {

15558 {CKA_CLASS, &class, sizeof(class)},

15559 {CKA_KEY_TYPE, &keyType, sizeof(keyType)},

15560 {CKA_TOKEN, &>true, sizeof(true)},

15561 {CKA_LABEL, label, sizeof(label)-1},

15562 {CKA_GOSTR3410_PARAMS, gostR3410params_oid,

15563 sizeof(gostR3410params_oid)},

15564 {CKA_GOSTR3411_PARAMS, gostR3411params_oid,

15565 sizeof(gostR3411params_oid)},

15566 {CKA_GOST28147_PARAMS, gost28147params_oid,

15567 sizeof(gost28147params_oid)},

15568 {CKA_VALUE, value, sizeof(value)}

15569 };

15570 **6.57.3 GOST R 34.10-2001 private key objects**

15571 GOST R 34.10-2001 private key objects (object class **CKO_PRIVATE_KEY**, key type

15572 **CKK_GOSTR3410**) hold GOST R 34.10-2001 private keys.

15573 The following table defines the GOST R 34.10-2001 private key object attributes, in addition to the

15574 common attributes defined for this object class:

15575 *Table 249, 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

Attribute	Data Type	Meaning
		CKK_GOSTR3411 must be specified with the same attribute CKA_OBJECT_ID
CKA_GOST28147_PARAMS ^{4,6,8}	Byte array	DER-encoding of the object identifier indicating the data object type of GOST 28147-89. When key is used the domain parameter object of key type CKK_GOST28147 must be specified with the same attribute CKA_OBJECT_ID . The attribute value may be omitted

15576 Refer to Table 13 for footnotes

15577 Note that when generating an GOST R 34.10-2001 private key, the GOST R 34.10-2001 domain
15578 parameters are *not* specified in the key's template. This is because GOST R 34.10-2001 private keys are
15579 only generated as part of an GOST R 34.10-2001 key *pair*, and the GOST R 34.10-2001 domain
15580 parameters for the pair are specified in the template for the GOST R 34.10-2001 public key.

15581 The following is a sample template for creating an GOST R 34.10-2001 private key object:

```

15582     CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
15583     CK_KEY_TYPE keyType = CKK_GOSTR3410;
15584     CK_UTF8CHAR label[] = "A GOST R34.10-2001 private key
15585         object";
15586     CK_BYTE subject[] = {...};
15587     CK_BYTE id[] = {123};
15588     CK_BYTE gostR3410params_oid[] =
15589         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
15590     CK_BYTE gostR3411params_oid[] =
15591         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1e, 0x00};
15592     CK_BYTE gost28147params_oid[] =
15593         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x1f, 0x00};
15594     CK_BYTE value[32] = {...};
15595     CK_BBOOL true = CK_TRUE;
15596     CK_ATTRIBUTE template[] = {
15597         {CKA_CLASS, &class, sizeof(class)},
15598         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
15599         {CKA_TOKEN, &true, sizeof(true)},
15600         {CKA_LABEL, label, sizeof(label)-1},
15601         {CKA_SUBJECT, subject, sizeof(subject)},
15602         {CKA_ID, id, sizeof(id)},
15603         {CKA_SENSITIVE, &true, sizeof(true)},
15604         {CKA_SIGN, &true, sizeof(true)},
15605         {CKA_GOSTR3410_PARAMS, gostR3410params_oid,
15606             sizeof(gostR3410params_oid)},
15607         {CKA_GOSTR3411_PARAMS, gostR3411params_oid,
15608             sizeof(gostR3411params_oid)},
15609         {CKA_GOST28147_PARAMS, gost28147params_oid,
15610             sizeof(gost28147params_oid)},

```

```
15611         {CKA_VALUE, value, sizeof(value)}
15612     };
15613
```

15614 **6.57.4 GOST R 34.10-2001 domain parameter objects**

15615 GOST R 34.10-2001 domain parameter objects (object class **CKO_DOMAIN_PARAMETERS**, key type
15616 **CKK_GOSTR3410**) hold GOST R 34.10-2001 domain parameters.

15617 The following table defines the GOST R 34.10-2001 domain parameter object attributes, in addition to the
15618 common attributes defined for this object class:

15619 *Table 250, 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

15620 ¹ Refer to Table 13 for footnotes

15621 For any particular token, there is no guarantee that a token supports domain parameters loading up
15622 and/or fetching out. Furthermore, applications, that make direct use of domain parameters objects, should
15623 take in account that **CKA_VALUE** attribute may be inaccessible.

15624 The following is a sample template for creating a GOST R 34.10-2001 domain parameter object:

```
15625     CK_OBJECT_CLASS class = CKO_DOMAIN_PARAMETERS;
15626     CK_KEY_TYPE keyType = CKK_GOSTR3410;
15627     CK_UTF8CHAR label[] = "A GOST R34.10-2001 cryptographic
15628         parameters object";
15629     CK_BYTE oid[] =
15630         {0x06, 0x07, 0x2a, 0x85, 0x03, 0x02, 0x02, 0x23, 0x00};
15631     CK_BYTE value[] = {
15632         0x30, 0x81, 0x90, 0x02, 0x01, 0x07, 0x02, 0x20, 0x5f, 0xbf, 0xf4, 0x98,
15633         0xaa, 0x93, 0x8c, 0xe7, 0x39, 0xb8, 0xe0, 0x22, 0xfb, 0xaf, 0xef, 0x40,
15634         0x56, 0x3f, 0x6e, 0x6a, 0x34, 0x72, 0xfc, 0x2a, 0x51, 0x4c, 0x0c, 0xe9,
15635         0xda, 0xe2, 0x3b, 0x7e, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00,
15636         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
15637         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
15638         0x00, 0x04, 0x31, 0x02, 0x21, 0x00, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00,
15639         0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x50, 0xfe,
15640         0x8a, 0x18, 0x92, 0x97, 0x61, 0x54, 0xc5, 0x9c, 0xfc, 0x19, 0x3a, 0xcc,
15641         0xf5, 0xb3, 0x02, 0x01, 0x02, 0x02, 0x20, 0x08, 0xe2, 0xa8, 0xa0, 0xe6,
15642         0x51, 0x47, 0xd4, 0xbd, 0x63, 0x16, 0x03, 0x0e, 0x16, 0xd1, 0x9c, 0x85,
15643         0xc9, 0x7f, 0x0a, 0x9c, 0xa2, 0x67, 0x12, 0x2b, 0x96, 0xab, 0xbc, 0xea,
15644         0x7e, 0x8f, 0xc8
15645     };
15646     CK_BBOOL true = CK_TRUE;
15647     CK_ATTRIBUTE template[] = {
15648         {CKA_CLASS, &class, sizeof(class)},
15649         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
15650         {CKA_TOKEN, &true, sizeof(true)},
```

```
15651         {CKA_LABEL, label, sizeof(label)-1},
15652         {CKA_OBJECT_ID, oid, sizeof(oid)},
15653         {CKA_VALUE, value, sizeof(value)}
15654     };
15655
```

15656 **6.57.5 GOST R 34.10-2001 mechanism parameters**

15657 **◆ CK_GOSTR3410_KEY_WRAP_PARAMS**

15658 **CK_GOSTR3410_KEY_WRAP_PARAMS** is a structure that provides the parameters to the
15659 **CKM_GOSTR3410_KEY_WRAP** mechanism. It is defined as follows:

```
15660     typedef struct CK_GOSTR3410_KEY_WRAP_PARAMS {
15661         CK_BYTE_PTR      pWrapOID;
15662         CK_ULONG          ulWrapOIDLen;
15663         CK_BYTE_PTR      pUKM;
15664         CK_ULONG          ulUKMLen;
15665         CK_OBJECT_HANDLE hKey;
15666     } CK_GOSTR3410_KEY_WRAP_PARAMS;
```

15667
15668 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

15669 **CK_GOSTR3410_KEY_WRAP_PARAMS_PTR** is a pointer to a
15670 **CK_GOSTR3410_KEY_WRAP_PARAMS**.

15671 **◆ CK_GOSTR3410_DERIVE_PARAMS**

15672 **CK_GOSTR3410_DERIVE_PARAMS** is a structure that provides the parameters to the
15673 **CKM_GOSTR3410_DERIVE** mechanism. It is defined as follows:

```
15674     typedef struct CK_GOSTR3410_DERIVE_PARAMS {  
15675         CK_EC_KDF_TYPE kdf;  
15676         CK_BYTE_PTR    pPublicData;  
15677         CK_ULONG       ulPublicDataLen;  
15678         CK_BYTE_PTR    pUKM;  
15679         CK_ULONG       ulUKMLen;  
15680     } CK_GOSTR3410_DERIVE_PARAMS;
```

15681
15682 The fields of the structure have the following meanings:

<i>kdf</i>	additional key diversification algorithm identifier. Possible values are CKD_NULL and CKD_CPDIVERSIFY_KDF . In case of CKD_NULL , the 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.
<i>pPublicData</i> ¹	pointer to data with public key of a receiver
<i>ulPublicDataLen</i>	length of data with public key of a receiver (must be 64)
<i>pUKM</i>	pointer to a UKM data
<i>ulUKMLen</i>	length of UKM data in bytes (must be 8)

15683
15684 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
15685 them is 32 bytes long and represented in little endian order.

15686 **CK_GOSTR3410_DERIVE_PARAMS_PTR** is a pointer to a **CK_GOSTR3410_DERIVE_PARAMS**.

15687 **6.57.6 GOST R 34.10-2001 key pair generation**

15688 The GOST R 34.10-2001 key pair generation mechanism, denoted
15689 **CKM_GOSTR3410_KEY_PAIR_GEN**, is a key pair generation mechanism for GOST R 34.10-2001.

15690 This mechanism does not have a parameter.

15691 The mechanism generates GOST R 34.10-2001 public/private key pairs with particular
15692 GOST R 34.10-2001 domain parameters, as specified in the **CKA_GOSTR3410_PARAMS**,
15693 **CKA_GOSTR3411_PARAMS**, and **CKA_GOST28147_PARAMS** attributes of the template for the public
15694 key. Note that **CKA_GOST28147_PARAMS** attribute may not be present in the template.

15695 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
15696 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_GOSTR3410_PARAMS**,
15697 **CKA_GOSTR3411_PARAMS**, **CKA_GOST28147_PARAMS** attributes to the new private key.

15698 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15699 are not used.

6.57.7 GOST R 34.10-2001 without hashing

The GOST R 34.10-2001 without hashing mechanism, denoted **CKM_GOSTR3410**, is a mechanism for single-part signatures and verification for GOST R 34.10-2001. (This mechanism corresponds only to the part of GOST R 34.10-2001 that processes the 32-bytes hash value; it does not compute the hash value.)

This mechanism does not have a parameter.

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 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 251, 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 252, 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.

15739 For unwrapping (**C_UnwrapKey**), the mechanism decrypts the wrapped key, and contributes the result as
15740 the **CKA_VALUE** attribute of the new key.
15741 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15742 are not used.

15743 **6.57.10 Common key derivation with assistance of GOST R 34.10-2001 keys**

15744 Common key derivation, denoted **CKM_GOSTR3410_DERIVE**, is a mechanism for key derivation with
15745 assistance of GOST R 34.10-2001 private and public keys. The key of the mechanism must be of object
15746 class **CKO_DOMAIN_PARAMETERS** and key type **CKK_GOSTR3410**. An algorithm for key derivation
15747 from [RFC 4357] (section 5.2) must be used.
15748 The mechanism contributes the result as the **CKA_VALUE** attribute of the new private key. All other
15749 attributes must be specified in a template for creating private key object.

15750 **6.58 ChaCha20**

15751 ChaCha20 is a secret-key stream cipher described in [CHACHA].

15752 *Table 253, ChaCha20 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_CHACHA20_KEY_GEN					✓			
CKM_CHACHA20	✓					✓		

15753

15754 **6.58.1 Definitions**

15755 This section defines the key type “**CKK_CHACHA20**” for type CK_KEY_TYPE as used in the
15756 **CKA_KEY_TYPE** attribute of key objects.

15757 Mechanisms:

15758 CKM_CHACHA20_KEY_GEN

15759 CKM_CHACHA20

15760 **6.58.2 ChaCha20 secret key objects**

15761 ChaCha20 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_CHACHA20**) hold
15762 ChaCha20 keys. The following table defines the ChaCha20 secret key object attributes, in addition to the
15763 common attributes defined for this object class:

15764 *Table 254, 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

15765 The following is a sample template for creating a ChaCha20 secret key object:

```
15766 CK_OBJECT_CLASS class = CKO_SECRET_KEY;  
15767 CK_KEY_TYPE keyType = CKK_CHACHA20;  
15768 CK_UTF8CHAR label[] = "A ChaCha20 secret key object";
```

```

15769 CK_BYTE value[32] = {...};
15770 CK_BBOOL true = CK_TRUE;
15771 CK_ATTRIBUTE template[] = {
15772     {CKA_CLASS, &class, sizeof(class)},
15773     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
15774     {CKA_TOKEN, &true, sizeof(true)},
15775     {CKA_LABEL, label, sizeof(label)-1},
15776     {CKA_ENCRYPT, &true, sizeof(true)},
15777     {CKA_VALUE, value, sizeof(value)}
15778 };

```

15779 **CKA_CHECK_VALUE**: The value of this attribute is derived from the key object by taking the first
15780 three bytes of the SHA-1 hash of the ChaCha20 secret key object's **CKA_VALUE** attribute.

15781 6.58.3 ChaCha20 mechanism parameters

15782 ♦ **CK_CHACHA20_PARAMS; CK_CHACHA20_PARAMS_PTR**

15783 **CK_CHACHA20_PARAMS** provides the parameters to the **CKM_CHACHA20** mechanism. It is defined
15784 as follows:

```

15785 typedef struct CK_CHACHA20_PARAMS {
15786     CK_BYTE_PTR    pBlockCounter;
15787     CK_ULONG        blockCounterBits;
15788     CK_BYTE_PTR    pNonce;
15789     CK_ULONG        ulNonceBits;
15790 } CK_CHACHA20_PARAMS;

```

15791 The fields of the structure have the following meanings:

15792	<i>pBlockCounter</i>	<i>pointer to block counter</i>
15793	<i>ulblockCounterBits</i>	<i>length of block counter in bits (can be either 32 or 64)</i>
15794	<i>pNonce</i>	<i>nonce (This should be never re-used with the same key.)</i>
15795	<i>ulNonceBits</i>	<i>length of nonce in bits (is 64 for original, 96 for IETF and 192 for xchacha20 variant)</i>
15796		

15797 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)
15798 it is necessary to address these blocks in random order, thus this counter is exposed here.

15799 **CK_CHACHA20_PARAMS_PTR** is a pointer to **CK_CHACHA20_PARAMS**.

15800 6.58.4 ChaCha20 key generation

15801 The ChaCha20 key generation mechanism, denoted **CKM_CHACHA20_KEY_GEN**, is a key generation
15802 mechanism for ChaCha20.

15803 It does not have a parameter.

15804 The mechanism generates ChaCha20 keys of 256 bits.

15805 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
15806 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key
15807 supports) may be specified in the template for the key, or else are assigned default initial values.

15808 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
15809 specify the supported range of key sizes in bytes. As a practical matter, the key size for ChaCha20 is
15810 fixed at 256 bits.

6.58.5 ChaCha20 mechanism

ChaCha20, denoted **CKM_CHACHA20**, is a mechanism for single and multiple-part encryption and decryption based on the ChaCha20 stream cipher. It comes in 3 variants, which only differ in the size and handling of their nonces, affecting the safety of using random nonces and the maximum size that can be encrypted safely.

Chacha20 has a parameter, **CK_CHACHA20_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 255, 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

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of ChaCha20 key sizes, in bits.

Table 256, 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.

Nonces must not ever be reused with the same key. However due to the birthday paradox the first two variants cannot guarantee that randomly generated nonces are never repeating. Thus the recommended way to handle this is to generate the first nonce randomly, then increase this for follow-up messages. Only the last (XChaCha20) has large enough nonces so that it is virtually impossible to trigger with randomly generated nonces the birthday paradox.

6.59 Salsa20

Salsa20 is a secret-key stream cipher described in **[SALSA]**.

Table 257, Salsa20 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
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 258, 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:

pBlockCounter *pointer to block counter (64 bits)*

15871 *pNonce* *nonce*

15872 *ulNonceBits* *size of the nonce in bits (64 for classic and 192 for XSalsa20)*

15873 The block counter is used to address 512 bit blocks in the stream. In certain settings (e.g. disk encryption)

15874 it is necessary to address these blocks in random order, thus this counter is exposed here.

15875 **CK_SALSA20_PARAMS_PTR** is a pointer to **CK_SALSA20_PARAMS**.

15876 **6.59.4 Salsa20 key generation**

15877 The Salsa20 key generation mechanism, denoted **CKM_SALSA20_KEY_GEN**, is a key generation

15878 mechanism for Salsa20.

15879 It does not have a parameter.

15880 The mechanism generates Salsa20 keys of 256 bits.

15881 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new

15882 key. Other attributes supported by the key type (specifically, the flags indicating which functions the key

15883 supports) may be specified in the template for the key, or else are assigned default initial values.

15884 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure

15885 specify the supported range of key sizes in bytes. As a practical matter, the key size for Salsa20 is fixed

15886 at 256 bits.

15887 **6.59.5 Salsa20 mechanism**

15888 Salsa20, denoted **CKM_SALSA20**, is a mechanism for single and multiple-part encryption and decryption

15889 based on the Salsa20 stream cipher. Salsa20 comes in two variants which only differ in the size and

15890 handling of their nonces, affecting the safety of using random nonces.

15891 Salsa20 has a parameter, **CK_SALSA20_PARAMS**, which indicates the nonce and initial block counter

15892 value.

15893 Constraints on key types and the length of input and output data are summarized in the following table:

15894 *Table 259, 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

15895 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure

15896 specify the supported range of ChaCha20 key sizes, in bits.

15897 *Table 260, 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.

15898 Nonces must not ever be reused with the same key. However due to the birthday paradox the original

15899 variant cannot guarantee that randomly generated nonces are never repeating. Thus the recommended

15900 way to handle this is to generate the first nonce randomly, then increase this for follow-up messages.

15901 Only the XSalsa20 has large enough nonces so that it is virtually impossible to trigger with randomly

15902 generated nonces the birthday paradox.

6.60 Poly1305

Poly1305 is a message authentication code designed by D.J Bernstein [POLY1305]. Poly1305 takes a 256 bit key and a message and produces a 128 bit tag that is used to verify the message.

Table 261, Poly1305 Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_POLY1305_KEY_GEN					✓			
CKM_POLY1305		✓						

6.60.1 Definitions

This section defines the key type “**CKK_POLY1305**” for type CK_KEY_TYPE as used in the **CKA_KEY_TYPE** attribute of key objects.

Mechanisms:

CKM_POLY1305_KEY_GEN

CKM_POLY1305

6.60.2 Poly1305 secret key objects

Poly1305 secret key objects (object class **CKO_SECRET_KEY**, key type **CKK_POLY1305**) hold Poly1305 keys. The following table defines the Poly1305 secret key object attributes, in addition to the common attributes defined for this object class:

Table 262, 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)}
};
```

15933 **6.60.3 Poly1305 mechanism**

15934 Poly1305, denoted **CKM_POLY1305**, is a mechanism for producing an output tag based on a 256 bit key
15935 and arbitrary length input.

15936 It has no parameters.

15937 Signatures (MACs) produced by this mechanism will be fixed at 128 bits in size.

15938 *Table 263, Poly1305: Key and Data Length*

Function	Key type	Data length	Signature Length
C_Sign	Poly1305	Any	128 bits
C_Verify	Poly1305	Any	128 bits

15939 **6.61 Chacha20/Poly1305 and Salsa20/Poly1305 Authenticated**
15940 **Encryption / Decryption**

15941 The stream ciphers Salsa20 and ChaCha20 are normally used in conjunction with the Poly1305
15942 authenticator, in such a construction they also provide Authenticated Encryption with Associated Data
15943 (AEAD). This section defines the combined mechanisms and their usage in an AEAD setting.

15944 *Table 264, Poly1305 Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_CHACHA20_POLY1305	✓							
CKM_SALSA20_POLY1305	✓							

15945 **6.61.1 Definitions**

15946 Mechanisms:

15947 CKM_CHACHA20_POLY1305

15948 CKM_SALSA20_POLY1305

15949 **6.61.2 Usage**

15950 Generic ChaCha20, Salsa20, Poly1305 modes are described in [CHACHA], [SALSA] and [POLY1305].
15951 To set up for ChaCha20/Poly1305 or Salsa20/Poly1305 use the following process. ChaCha20/Poly1305
15952 and Salsa20/Poly1305 both use CK_SALSA20_CHACHA20_POLY1305_PARAMS for Encrypt, Decrypt
15953 and CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS for MessageEncrypt, and MessageDecrypt.
15954 Encrypt:

- 15955 • Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of Chacha20
15956 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)
- 15957 • Set the Nonce data *pNonce* in the parameter block.
- 15958 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
15959 *ulAADLen* is 0.
- 15960 • Call **C_EncryptInit** for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
15961 mechanism with parameters and key *K*.

15962 • Call **C_Encrypt**, or **C_EncryptUpdate**^{*10} **C_EncryptFinal**, for the plaintext obtaining ciphertext
15963 and authentication tag output.

15964 Decrypt:

15965 • Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of ChaCha20
15966 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

15967 • Set the Nonce data *pNonce* in the parameter block.

15968 • Set the AAD data *pAAD* and size *ulAADLen* in the parameter block. *pAAD* may be NULL if
15969

15970 • Call **C_DecryptInit** for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
15971 mechanism with parameters and key *K*.

15972 • Call **C_Decrypt**, or **C_DecryptUpdate**^{*11} **C_DecryptFinal**, for the ciphertext, including the
15973 appended tag, obtaining plaintext output. Note: since **CKM_CHACHA20_POLY1305** and
15974 **CKM_SALSA20_POLY1305** are AEAD ciphers, no data should be returned until **C_Decrypt** or
15975 **C_DecryptFinal**.

15976 MessageEncrypt:

15977 • Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of ChaCha20
15978 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

15979 • Set the Nonce data *pNonce* in the parameter block.

15980 • Set *pTag* to hold the tag data returned from **C_EncryptMessage** or the final
15981 **C_EncryptMessageNext**.

15982 • Call **C_MessageEncryptInit** for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
15983 mechanism with key *K*.

15984 • Call **C_EncryptMessage**, or **C_EncryptMessageBegin** followed by **C_EncryptMessageNext**^{*11}.
15985 The mechanism parameter is passed to all three of these functions.

15986 • Call **C_MessageEncryptFinal** to close the message decryption.

15987 MessageDecrypt:

15988 • Set the Nonce length *ulNonceLen* in the parameter block. (this affects which variant of ChaCha20
15989 will be used: 64 bits → original, 96 bits → IETF, 192 bits → XChaCha20)

15990 • Set the Nonce data *pNonce* in the parameter block.

15991 • Set the tag data *pTag* in the parameter block before **C_DecryptMessage** or the final
15992 **C_DecryptMessageNext**

15993 • Call **C_MessageDecryptInit** for **CKM_CHACHA20_POLY1305** or **CKM_SALSA20_POLY1305**
15994 mechanism with key *K*.

15995 • Call **C_DecryptMessage**, or **C_DecryptMessageBegin** followed by **C_DecryptMessageNext**^{*12}.
15996 The mechanism parameter is passed to all three of these functions.

15997 • Call **C_MessageDecryptFinal** to close the message decryption

15998

10 "*" indicates 0 or more calls may be made as required

11 "*" indicates 0 or more calls may be made as required

12 "*" indicates 0 or more calls may be made as required

15999 *ulNonceLen* is the length of the nonce in bits.

16000 In Encrypt and Decrypt the tag is appended to the ciphertext. In MessageEncrypt the tag is returned in
 16001 the pTag field of CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS. In MessageDecrypt the tag is
 16002 provided by the pTag field of CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS. The application
 16003 must provide 16 bytes of space for the tag.

16004 The key type for *K* must be compatible with **CKM_CHACHA20** or **CKM_SALSA20** respectively and the
 16005 **C_EncryptInit/C_DecryptInit** calls shall behave, with respect to *K*, as if they were called directly with
 16006 **CKM_CHACHA20** or **CKM_SALSA20**, *K* and NULL parameters.

16007 Unlike the atomic Salsa20/ChaCha20 mechanism the AEAD mechanism based on them does not expose
 16008 the block counter, as the AEAD construction is based on a message metaphor in which random access is
 16009 not needed.

16010 6.61.3 ChaCha20/Poly1305 and Salsa20/Poly1305 Mechanism parameters

16011 ♦ **CK_SALSA20_CHACHA20_POLY1305_PARAMS;** 16012 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR**

16013 **CK_SALSA20_CHACHA20_POLY1305_PARAMS** is a structure that provides the parameters to the
 16014 **CKM_CHACHA20_POLY1305** and **CKM_SALSA20_POLY1305** mechanisms. It is defined as follows:

```
16015 typedef struct CK_SALSA20_CHACHA20_POLY1305_PARAMS {
16016     CK_BYTE_PTR  pNonce;
16017     CK_ULONG     ulNonceLen;
16018     CK_BYTE_PTR  pAAD;
16019     CK_ULONG     ulAADLen;
16020 } CK_SALSA20_CHACHA20_POLY1305_PARAMS;
```

16021 The fields of the structure have the following meanings:

16022	<i>pNonce</i>	<i>nonce (This should be never re-used with the same key.)</i>
16023	<i>ulNonceLen</i>	<i>length of nonce in bits (is 64 for original, 96 for IETF (only for</i>
16024		<i>chacha20) and 192 for xchacha20/xsalsa20 variant)</i>
16025	<i>pAAD</i>	<i>pointer to additional authentication data. This data is authenticated</i>
16026		<i>but not encrypted.</i>
16027	<i>ulAADLen</i>	<i>length of pAAD in bytes.</i>

16028 **CK_SALSA20_CHACHA20_POLY1305_PARAMS_PTR** is a pointer to a
 16029 **CK_SALSA20_CHACHA20_POLY1305_PARAMS**.

16030 ♦ **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;** 16031 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR**

16032 **CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS** is a structure that provides the parameters to the
 16033 **CKM_CHACHA20_POLY1305** mechanism. It is defined as follows:

```
16034 typedef struct CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS {
16035     CK_BYTE_PTR  pNonce;
16036     CK_ULONG     ulNonceLen;
16037     CK_BYTE_PTR  pTag;
16038 } CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS;
```

16039 The fields of the structure have the following meanings:

16040	<i>pNonce</i>	<i>pointer to nonce</i>
-------	---------------	-------------------------

16041160421604316044160451604616047

ulNonceLen

pTag

length of nonce in bits. The length of the influences which variant of the ChaCha20 will be used (64 original, 96 IETF(only for ChaCha20), 192 XChaCha20/XSalsa20)

location of the authentication tag which is returned on MessageEncrypt, and provided on MessageDecrypt.

CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS_PTR is a pointer to a CK_SALSA20_CHACHA20_POLY1305_MSG_PARAMS.

16048

6.62 HKDF Mechanisms

16049

Details for HKDF key derivation mechanisms can be found in [RFC 5869].

1605016051

Table 265, HKDF Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_HKDF_DERIVE							✓	
CKM_HKDF_DATA							✓	
CKM_HKDF_KEY_GEN					✓			

16052

6.62.1 Definitions

16053160541605516056160571605816059

Mechanisms:
CKM_HKDF_DERIVE
CKM_HKDF_DATA
CKM_HKDF_KEY_GEN

Key Types:
CKK_HKDF

16060

6.62.2 HKDF mechanism parameters

160611606216063160641606516066160671606816069160701607116072

♦ 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;

```

16073         CK_ULONG ulInfoLen;
16074     } CK_HKDF_PARAMS;
16075

```

16076 The fields of the structure have the following meanings:

16077	bExtract	execute the extract portion of HKDF.
16078	bExpand	execute the expand portion of HKDF.
16079	prfHashMechanism	base hash used for the HMAC in the underlying HKDF operation.
16080	ulSaltType	specifies how the salt for the extract portion of the KDF is supplied.
16081		CKF_HKDF_SALT_NULL no salt is supplied.
16082		CKF_HKDF_SALT_DATA salt is supplied as a data in pSalt with
16083		length ulSaltLen.
16084		CKF_HKDF_SALT_KEY salt is supplied as a key in hSaltKey.
16085	pSalt	pointer to the salt.
16086	ulSaltLen	length of the salt pointed to in pSalt.
16087	hSaltKey	object handle to the salt key.
16088	pInfo	info string for the expand stage.
16089	ulInfoLen	length of the info string for the expand stage.

16090

16091 **CK_HKDF_PARAMS_PTR** is a pointer to a **CK_HKDF_PARAMS**.

16092 6.62.3 HKDF derive

16093 HKDF derivation implements the HKDF as specified in [RFC 5869]. The two booleans bExtract and
16094 bExpand control whether the extract section of the HKDF or the expand section of the HKDF is in use.

16095 It has a parameter, a **CK_HKDF_PARAMS** structure, which allows for the passing of the salt and or the
16096 expansion info. The structure contains the bools *bExtract* and *bExpand* which control whether the extract
16097 or expand portions of the HKDF is to be used. This structure is defined in section 6.62.2.

16098 The input key must be of type **CKK_HKDF** or **CKK_GENERIC_SECRET** and the length must be the size
16099 of the underlying hash function specified in *prfHashMechanism*. The exception is a data object which has
16100 the same size as the underlying hash function, and which may be supplied as an input key. In this case
16101 bExtract should be true and non-null salt should be supplied.

16102 Either *bExtract* or *bExpand* must be set to true. If they are both set to true, input key is first extracted then
16103 expanded. The salt is used in the extraction stage. If bExtract is set to true and no salt is given, a 'zero'
16104 salt (salt whose length is the same as the underlying hash and values all set to zero) is used as specified
16105 by the RFC. If bExpand is set to true, **CKA_VALUE_LEN** should be set to the desired key length. If it is
16106 false **CKA_VALUE_LEN** may be set to the length of the hash, but that is not necessary as the
16107 mechanism will supply this value. The salt should be ignored if *bExtract* is false. The *pInfo* should be
16108 ignored if *bExpand* is set to false.

16109 The mechanism also contributes the **CKA_CLASS**, and **CKA_VALUE** attributes to the new key. Other
16110 attributes may be specified in the template, or else are assigned default values.

16111 The template sent along with this mechanism during a **C_DeriveKey** call may indicate that the object
16112 class is **CKO_SECRET_KEY**. However, since these facts are all implicit in the mechanism, there is no
16113 need to specify any of them.

16114 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 *plnfo* portion of the CK_HKDF_PARAMS. Tokens may reject requests based on the *plnfo* values. Allowed *plnfo* 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.

Table 266, *CKM_NULL Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_NULL	✓	✓	✓	✓		✓	✓	

6.63.1 Definitions

Mechanisms:
 CKM_NULL

6.63.2 CKM_NULL mechanism parameters

CKM_NULL does not have a parameter.

When used for encrypting / decrypting data, the input data is copied unchanged to the output data.

When used for signing, the input data is copied to the signature. When used for signature verification, it compares the input data and the signature, and returns **CKR_OK** (indicating that both are identical) or **CKR_SIGNATURE_INVALID**.

16150 When used for digesting data, the input data is copied to the message digest.

16151 When used for wrapping a private or secret key object, the wrapped key will be identical to the key to be

16152 wrapped. When used for unwrapping, a new object with the same value as the wrapped key will be

16153 created.

16154 When used for deriving a key, the derived key has the same value as the base key.

16155 When used for key encapsulation, the encapsulated key will be identical to the secret key to be

16156 encapsulated. When used for key decapsulation, a new object with the same value as the encapsulated

16157 key will be created.

16158 **6.64 IKE Mechanisms**

16159

16160 *Table 267, IKE Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_IKE2_PRF_PLUS_DERIVE							✓	
CKM_IKE_PRF_DERIVE							✓	
CKM_IKE1_PRF_DERIVE							✓	
CKM_IKE1_EXTENDED_DERIVE							✓	

16161

16162 **6.64.1 Definitions**

16163 Mechanisms:

- 16164 CKM_IKE2_PRF_PLUS_DERIVE
- 16165 CKM_IKE_PRF_DERIVE
- 16166 CKM_IKE1_PRF_DERIVE
- 16167 CKM_IKE1_EXTENDED_DERIVE

16168

16169 **6.64.2 IKE mechanism parameters**

- 16170 ♦ **CK_IKE2_PRF_PLUS_DERIVE_PARAMS;**
- 16171 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR**

16172 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS** is a structure that provides the parameters to the

16173 **CKM_IKE2_PRF_PLUS_DERIVE** mechanism. It is defined as follows:

```
16174 typedef struct CK_IKE2_PRF_PLUS_DERIVE_PARAMS {
16175     CK_MECHANISM_TYPE prfMechanism;
16176     CK_BBOOL bHasSeedKey;
16177     CK_OBJECT_HANDLE hSeedKey;
16178     CK_BYTE_PTR pSeedData;
16179     CK_ULONG ulSeedDataLen;
16180 } CK_IKE2_PRF_PLUS_DERIVE_PARAMS;
```

16181

16182 The fields of the structure have the following meanings:

16183	prfMechanism	underlying MAC mechanism used to generate the prf
16184	bHasSeedKey	hSeed key is present
16185	hSeedKey	optional seed from key
16186	pSeedData	optional seed from data
16187	ulSeedDataLen	length of optional seed data. If no seed data is present this value
16188		is 0

16189 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS_PTR** is a pointer to a
 16190 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS**.

16191

16192 ♦ **CK_IKE_PRF_DERIVE_PARAMS;**
 16193 **CK_IKE_PRF_DERIVE_PARAMS_PTR**

16194 **CK_IKE_PRF_DERIVE_PARAMS** is a structure that provides the parameters to the
 16195 **CKM_IKE_PRF_DERIVE** mechanism. It is defined as follows:

16196

```
16197     typedef struct CK_IKE_PRF_DERIVE_PARAMS {
16198         CK_MECHANISM_TYPE  prfMechanism;
16199         CK_BBOOL           bDataAsKey;
16200         CK_BBOOL           bRekey;
16201         CK_BYTE_PTR        pNi;
16202         CK_ULONG           ulNiLen;
16203         CK_BYTE_PTR        pNr;
16204         CK_ULONG           ulNrLen;
16205         CK_OBJECT_HANDLE    hNewKey;
16206     } CK_IKE_PRF_DERIVE_PARAMS;
```

16207

16208 The fields of the structure have the following meanings:

16209	prfMechanism	underlying MAC mechanism used to generate the prf
16210	bDataAsKey	Ni Nr is used as the key for the prf rather than baseKey
16211	bRekey	rekey operation. hNewKey must be present
16212	pNi	Ni value
16213	ulNiLen	length of Ni
16214	pNr	Nr value
16215	ulNrLen	length of Nr
16216	hNewKey	New key value to drive the rekey.

16217 **CK_IKE_PRF_DERIVE_PARAMS_PTR** is a pointer to a **CK_IKE_PRF_DERIVE_PARAMS**.

16218

16219 ♦ **CK_IKE1_PRF_DERIVE_PARAMS;**
 16220 **CK_IKE1_PRF_DERIVE_PARAMS_PTR**

16221 **CK_IKE1_PRF_DERIVE_PARAMS** is a structure that provides the parameters to the
 16222 **CKM_IKE1_PRF_DERIVE** mechanism. It is defined as follows:

```
16223     typedef struct CK_IKE1_PRF_DERIVE_PARAMS {
16224         CK_MECHANISM_TYPE   prfMechanism;
16225         CK_BBOOL             bHasPrevKey;
16226         CK_OBJECT_HANDLE     hKeygxy;
16227         CK_OBJECT_HANDLE     hPrevKey;
16228         CK_BYTE_PTR          pCKYi;
16229         CK_ULONG              ulCKYiLen;
16230         CK_BYTE_PTR          pCKYr;
16231         CK_ULONG              ulCKYrLen;
16232         CK_BYTE               keyNumber;
16233     } CK_IKE1_PRF_DERIVE_PARAMS;
```

16234
16235 The fields of the structure have the following meanings:

16236	prfMechanism	underlying MAC mechanism used to generate the prf
16237	bHasPrevkey	hPrevKey is present
16238	hKeygxy	handle to the exchanged g ^{xy} key
16239	hPrevKey	handle to the previously derived key
16240	pCKYi	CKYi value
16241	ulCKYiLen	length of CKYi
16242	pCKYr	CKYr value
16243	ulCKYrLen	length of CKYr
16244	keyNumber	unique number for this key derivation

16245 **CK_IKE1_PRF_DERIVE_PARAMS_PTR** is a pointer to a **CK_IKE1_PRF_DERIVE_PARAMS**.

16246

16247 ♦ **CK_IKE1_EXTENDED_DERIVE_PARAMS;**
16248 **CK_IKE1_EXTENDED_DERIVE_PARAMS_PTR**

16249 **CK_IKE1_EXTENDED_DERIVE_PARAMS** is a structure that provides the parameters to the
16250 **CKM_IKE1_EXTENDED_DERIVE** mechanism. It is defined as follows:

16251

```
16252     typedef struct CK_IKE1_EXTENDED_DERIVE_PARAMS {
16253         CK_MECHANISM_TYPE   prfMechanism;
16254         CK_BBOOL             bHasKeygxy;
16255         CK_OBJECT_HANDLE     hKeygxy;
16256         CK_BYTE_PTR          pExtraData;
16257         CK_ULONG              ulExtraDataLen;
16258     } CK_IKE1_EXTENDED_DERIVE_PARAMS;
```

16259 The fields of the structure have the following meanings:

16260	prfMechanism	underlying MAC mechanism used to generate the prf
16261	bHasKeygxy	hKeygxy key is present
16262	hKeygxy	optional key g ^{xy}
16263	pExtraData	optional extra data

16308 for outKey = SKEYID_e, hPrevKey= SKEYID_a, key_number = 2
16309 If **CKA_VALUE_LEN** is not specified, the resulting key will be the length of the prf. If **CKA_VALUE_LEN**
16310 is greater then the prf, **CKR_KEY_SIZE_RANGE** is returned. If it is less the key is truncated taking the
16311 left most bytes. The value **CKA_KEY_TYPE** must be specified in the template or
16312 **CKR_TEMPLATE_INCOMPLETE** is returned.
16313

16314 6.64.5 IKEv2 PRF PLUS DERIVE

16315 The IKEv2 PRF PLUS Derive mechanism denoted **CKM_IKE2_PRF_PLUS_DERIVE** is used in IPSEC
16316 IKEv2 to derive various additional keys from the initial SKEYSEED. It takes a
16317 **CK_IKE2_PRF_PLUS_DERIVE_PARAMS** as a mechanism parameter. SKEYSEED is the base key
16318 passed into **C_DeriveKey**. The key type of *baseKey* must be the key type of the underlying prf. This
16319 mechanism uses the base key and a feedback version of the prf to generate a single key with sufficient
16320 bytes to cover all additional keys. The application will then use **CKM_EXTRACT_KEY_FROM_KEY**
16321 several times to pull out the various keys. **CKA_VALUE_LEN** must be set in the template and its value
16322 must not be bigger than 255 times the size of the prf function output or **CKR_KEY_SIZE_RANGE** will be
16323 returned. If **CKA_KEY_TYPE** is not specified, the output key type will be **CKK_GENERIC_SECRET**.
16324

16325 This mechanism derives a key with a **CKA_VALUE** of (from [RFC 5996]):
16326

16327 $\text{prfplus} = T1 \mid T2 \mid T3 \mid T4 \mid \dots Tn$

16328 where:

16329 $T1 = \text{prf}(K, S \mid 0x01)$

16330 $T2 = \text{prf}(K, T1 \mid S \mid 0x02)$

16331 $T3 = \text{prf}(K, T3 \mid S \mid 0x03)$

16332 $T4 = \text{prf}(K, T4 \mid S \mid 0x04)$

16333 .

16334 $Tn = \text{prf}(K, T(n-1) \mid n)$

16335 $K = \text{baseKey}, S = \text{valueOf}(hSeedKey) \mid pSeedData$

16336 6.64.6 IKEv1 Extended Derive

16337 The IKE Extended Derive mechanism denoted **CKM_IKE1_EXTENDED_DERIVE** is used in IPSEC
16338 IKEv1 to derive longer keys than **CKM_IKE1_EXTENDED_DERIVE** can from the initial SKEYID. It is
16339 used to support [RFC 2409] appendix B and section 5.5 (Quick Mode). It takes a
16340 **CK_IKE1_EXTENDED_DERIVE_PARAMS** as a mechanism parameter. SKEYID is the base key passed
16341 into **C_DeriveKey**. **CKA_VALUE_LEN** must be set in the template and its value must not be bigger than
16342 255 times the size of the prf function output or **CKR_KEY_SIZE_RANGE** will be returned. If
16343 **CKA_KEY_TYPE** is not specified, the output key type will be **CKK_GENERIC_SECRET**. The key type of
16344 SKEYID must be the key type of the prf, and the key type of *hKeygxy* (if present) must be
16345 **CKK_GENERIC_SECRET**.
16346

16347 This mechanism derives a key with **CKA_VALUE** (from [RFC 2409] appendix B and section 5.5):
16348

16348 $Ka = K1 \mid K2 \mid K3 \mid K4 \mid \dots Kn$

16349 where:

16350 $K1 = \text{prf}(K, \text{valueOf}(hKeygxy) \mid pExtraData)$ or $\text{prf}(K, 0x00)$ if *bHashKeygxy* is FALSE and *ulExtraData*
16351 is 0

16352 $K2 = \text{prf}(K, K1 \mid \text{valueOf}(hKeygxy) \mid pExtraData)$

16353 $K3 = \text{prf}(K, K2 \mid \text{valueOf}(hKeygxy) \mid pExtraData)$

16354 K4 = prf(K, K3|valueOf(hKeygxy)|pExtraData)
16355 .
16356 Kn = prf(K, K(n-1)|valueOf(hKeygxy)|pExtraData)
16357 K = baseKey

16358
16359 If **CKA_VALUE_LEN** is less then or equal to the prf length and bHasKeygxy is CK_FALSE, then the new
16360 key is simply the base key truncated to **CKA_VALUE_LEN** (specified in RFC 2409 appendix B).
16361 Otherwise the prf is executed and the derived keys value is **CKA_VALUE_LEN** bytes of the resulting prf.

16362 **6.65 HSS**

16363 HSS is a mechanism for single-part signatures and verification, following the digital signature algorithm
16364 defined in [RFC 8554] and [NIST SP800-208].

16365
16366 *Table 268, HSS Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_HSS_KEY_PAIR_GEN					✓			
CKM_HSS		✓ ¹						

16367 1 Single-part operations only
16368

16369 **6.65.1 Definitions**

16370 This section defines the key type **CKK_HSS** for type **CK_KEY_TYPE** as used in the **CKA_KEY_TYPE**
16371 attribute of key objects and domain parameter objects.

16372 Mechanisms:
16373 CKM_HSS_KEY_PAIR_GEN
16374 CKM_HSS

16375 **6.65.2 HSS public key objects**

16376 HSS public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_HSS**) hold HSS public keys.
16377 The following table defines the HSS public key object attributes, in addition to the common attributes
16378 defined for this object class:

16379 *Table 269, 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 13 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 270, 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 <i>ulValueLen</i> 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 <i>ulValueLen</i> 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) $2^{32}-1$

16405 Refer to Table 13 for footnotes

16406

16407 The encodings for **CKA_HSS_LMOTS_TYPES** and **CKA_HSS_LMS_TYPES** are defined in [RFC 8554]
16408 and [NIST SP800-208].

16409

16410 The following is a sample template for creating an LMS private key object:

16411

```

16412 CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
16413 CK_KEY_TYPE keyType = CKK_HSS;
16414 CK_UTF8CHAR label[] = "An HSS private key object";
16415 CK_ULONG hssLevels = 123;
16416 CK_ULONG lmsTypes[] = {123,...};
16417 CK_ULONG lmotsTypes[] = {123,...};
16418 CK_BYTE value[] = {...};
16419 CK_BBOOL true = CK_TRUE;
16420 CK_BBOOL false = CK_FALSE;
16421 CK_ATTRIBUTE template[] = {
```

```

16422     {CKA_CLASS, &keyClass, sizeof(keyClass)},
16423     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
16424     {CKA_TOKEN, &true, sizeof(true)},
16425     {CKA_LABEL, label, sizeof(label)-1},
16426     {CKA_SENSITIVE, &true, sizeof(true)},
16427     {CKA_EXTRACTABLE, &false, sizeof(true)},
16428     {CKA_HSS_LEVELS, &hssLevels, sizeof(hssLevels)},
16429     {CKA_HSS_LMS_TYPES, lmsTypes, sizeof(lmsTypes)},
16430     {CKA_HSS_LMOTS_TYPES, lmotsTypes, sizeof(lmotsTypes)},
16431     {CKA_VALUE, value, sizeof(value)},
16432     {CKA_SIGN, &true, sizeof(true)}
16433 };
16434

```

16435 **CKA_SENSITIVE** MUST be true, **CKA_EXTRACTABLE** MUST be false, and **CKA_COPYABLE** MUST
16436 be false for this key.

16437 6.65.4 HSS key pair generation

16438 The HSS key pair generation mechanism, denoted **CKM_HSS_KEY_PAIR_GEN**, is a key pair generation
16439 mechanism for HSS.

16440 This mechanism does not have a parameter.

16441 The mechanism generates HSS public/private key pairs for the scheme specified by the
16442 **CKA_HSS_LEVELS**, **CKA_HSS_LMS_TYPES**, and **CKA_HSS_LMOTS_TYPES** attributes of the
16443 template for the private key.

16444 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_HSS_LEVELS**,
16445 **CKA_HSS_LMS_TYPE**, **CKA_HSS_LMOTS_TYPE**, and **CKA_VALUE** attributes to the new public key
16446 and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_HSS_KEYS_REMAINING** attributes
16447 to the new private key.

16448 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
16449 are not used and must be set to 0.

16450 6.65.5 HSS without hashing

16451 The HSS without hashing mechanism, denoted **CKM_HSS**, is a mechanism for single-part signatures and
16452 verification for HSS. (This mechanism corresponds only to the part of LMS that processes the hash value,
16453 which may be of any length; it does not compute the hash value.)

16454 This mechanism does not have a parameter.

16455 For the purposes of these mechanisms, an HSS signature is a byte string with length depending on
16456 **CKA_HSS_LEVELS**, **CKA_HSS_LMS_TYPES**, **CKA_HSS_LMOTS_TYPES** as described in the
16457 following table.

16458 *Table 271, 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

¹ Single-part operations only.
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 Merkle trees.

For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure are not used and must be set to 0.
If the number of signatures is exhausted, **CKR_KEY_EXHAUSTED** will be returned.

6.66 XMSS and XMSS^{MT}

XMSS and XMSS^{MT} are mechanisms for single-part signatures and verification, following the digital signature algorithm defined in [RFC 8391].

Table 272, XMSS and XMSS^{MT} Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encapsulate & Decapsulate
CKM_XMSS_KEY_PAIR_GEN					✓			
CKM_XMSS		✓ ¹						
CKM_XMSSMT_KEY_PAIR_GEN					✓			
CKM_XMSSMT		✓ ¹						

¹ Single-part operations only

6.66.1 Definitions

This section defines the key type **CKK_XMSS** and **CKK_XMSSMT** for type **CK_KEY_TYPE** as used in the **CKA_KEY_TYPE** attribute of key objects and domain parameter objects.
Mechanisms:
 CKM_XMSS_KEY_PAIR_GEN
 CKM_XMSS
 CKM_XMSSMT_KEY_PAIR_GEN
 CKM_XMSSMT

6.66.2 XMSS public key objects

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

16486 Table 273, XMSS Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,3}	CK_XMSS_PARAMETER_SET_TYPE	XMSS parameter set as defined below.
CKA_VALUE ^{1,4}	Byte array	4+2*N bytes; a 4 byte integer in big-endian order encoding an <i>algorithm OID</i> , an N byte string (the Merkle tree root hash value), an N byte string (the public SEED).

16487 Refer to Table 13 for footnotes

16488

16489 typedef CK_ULONG CK_XMSS_PARAMETER_SET_TYPE;

16490 where CK_XMSS_PARAMETER_SET_TYPE is the numeric identifier of the XMSS parameter set defined
16491 in [NIST SP800-208].

16492

16493 The following is a sample template for creating an XMSS public key object:

```
16494 CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
16495 CK_KEY_TYPE keyType = CKK_XMSS;
16496 CK_UTF8CHAR label[] = "An XMSS public key object";
16497 CK_XMSS_PARAMETER_SET_TYPE xmss_param = 0x01;
16498 CK_BYTE value[] = {...};
16499 CK_BBOOL true = CK_TRUE;
16500 CK_BBOOL false = CK_FALSE;
16501
16502 CK_ATTRIBUTE template[] = {
16503     {CKA_CLASS, &keyClass, sizeof(keyClass)},
16504     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
16505     {CKA_TOKEN, &>false, sizeof(false)},
16506     {CKA_LABEL, label, sizeof(label)-1},
16507     {CKA_PARAMETER_SET, &xmss_param, sizeof(xmss_param)},
16508     {CKA_VALUE, value, sizeof(value)},
16509     {CKA_VERIFY, &>true, sizeof(true)}
16510 };
```

16511 **6.66.3 XMSS^{MT} public key objects**

16512 XMSS^{MT} public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_XMSSMT**) hold XMSS^{MT}
16513 public keys.

16514 The following table defines the XMSS^{MT} public key object attributes, in addition to the common attributes
16515 defined for this object class:

16516 Table 274, XMSS^{MT} Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,3}	CK_XMSS MT_PARA METER_SE T_TYPE	XMSS ^{MT} parameter set as defined below.
CKA_VALUE ^{1,4}	Byte array	4+2*N bytes; a 4 byte integer in big-endian order encoding an <i>algorithm OID</i> , an N byte string (the Merkle tree root hash value), an N byte string (the public SEED).

16517 Refer to Table 13 for footnotes

16518

16519 `typedef CK_ULONG CK_XMSSMT_PARAMETER_SET_TYPE;`

16520 where CK_XMSSMT_PARAMETER_SET_TYPE is the numeric identifier of the XMSS^{MT} parameter set
16521 defined in [NIST SP800-208].

16522

16523 The following is a sample template for creating an XMSS^{MT} public key object:

```
16524 CK_OBJECT_CLASS keyClass = CKO_PUBLIC_KEY;
16525 CK_KEY_TYPE keyType = CKK_XMSSMT;
16526 CK_UTF8CHAR label[] = "An XMSSMT public key object";
16527 CK_XMSSMT_PARAMETER_SET_TYPE xmss_mt_param = 0x01;
16528 CK_BYTE value[] = {...};
16529 CK_BBOOL true = CK_TRUE;
16530 CK_BBOOL false = CK_FALSE;
16531
16532 CK_ATTRIBUTE template[] = {
16533     {CKA_CLASS, &keyClass, sizeof(keyClass)},
16534     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
16535     {CKA_TOKEN, &>false, sizeof(false)},
16536     {CKA_LABEL, label, sizeof(label)-1},
16537     {CKA_PARAMETER_SET, &xmss_mt_param, sizeof(xmss_mt_param)},
16538     {CKA_VALUE, value, sizeof(value)},
16539     {CKA_VERIFY, &>true, sizeof(true)}
16540 };
```

16541 **6.66.4 XMSS private key objects**

16542 XMSS private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_XMSS**) hold XMSS private
16543 keys.

16544 The following table defines the XMSS private key object attributes, in addition to the common attributes
16545 defined for this object class:

16546 Table 275, XMSS Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,4,6}	CK_XMSS_PARAMETER_SET_TYPE	Numeric identifier of the XMSS parameter set as defined in section 6.66.2.
CKA_VALUE ^{1,4,6,7}	Byte array	Vendor defined. Note that exporting this value is dangerous as it would allow key reuse.

Refer to Table 13 for footnotes

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

```

CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_XMSS;
CK_UTF8CHAR label[] = "An XMSS private key object";
CK_XMSS_PARAMETER_SET_TYPE xmss_param = 0x01;
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, &true, sizeof(true)},
    {CKA_LABEL, label, sizeof(label)-1},
    {CKA_SENSITIVE, &true, sizeof(true)},
    {CKA_EXTRACTABLE, &false, sizeof(true)},
    {CKA_PARAMETER_SET, &xmss_param, sizeof(xmss_param)},
    {CKA_VALUE, value, sizeof(value)}
    {CKA_SIGN, &true, sizeof(true)}
};

```

CKA_SENSITIVE MUST be true and **CKA_EXTRACTABLE** MUST be false for this key.

6.66.5 XMSS^{MT} private key objects

XMSS^{MT} private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_XMSSMT**) hold XMSS^{MT} private keys.

The following table defines the XMSS^{MT} private key object attributes, in addition to the common attributes defined for this object class:

Table 276, XMSS^{MT} Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,4,6}	CK_XMSSMT_PARAMETER_SET_TYPE	Numeric identifier of the XMSS ^{MT} parameter set as defined in section 6.66.3.
CKA_VALUE ^{1,4,6,7}	Byte array	Vendor defined. Note that exporting this value is dangerous as it would allow key reuse.

Refer to Table 13 for footnotes

```

16576
16577 The following is a sample template for creating an XMSSMT private key object:
16578     CK_OBJECT_CLASS keyClass = CKO_PRIVATE_KEY;
16579     CK_KEY_TYPE keyType = CKK_XMSS;
16580     CK_UTF8CHAR label[] = "An XMSSMT private key object";
16581     CK_XMSSMT_PARAMETER_SET_TYPE xmss_mt_param = 0x01;
16582     CK_BYTE value[] = {...};
16583     CK_BBOOL true = CK_TRUE;
16584     CK_BBOOL false = CK_FALSE;
16585     CK_ATTRIBUTE template[] = {
16586         {CKA_CLASS, &keyClass, sizeof(keyClass)},
16587         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
16588         {CKA_TOKEN, &true, sizeof(true)},
16589         {CKA_LABEL, label, sizeof(label)-1},
16590         {CKA_SENSITIVE, &true, sizeof(true)},
16591         {CKA_EXTRACTABLE, &false, sizeof(true)},
16592         {CKA_PARAMETER_SET, &xmss_mt_param, sizeof(xmss_mt_param)},
16593         {CKA_VALUE, value, sizeof(value)}
16594         {CKA_SIGN, &true, sizeof(true)}
16595     };

```

16596 **CKA_SENSITIVE** MUST be true and **CKA_EXTRACTABLE** MUST be false for this key.

16597 6.66.6 XMSS key pair generation

16598 The XMSS key pair generation mechanism, denoted **CKM_XMSS_KEY_PAIR_GEN**, is a key pair
16599 generation mechanism for XMSS.

16600 This mechanism does not have a parameter.

16601 The mechanism generates XMSS public/private key pairs using an *oid*, as specified in the
16602 **CKA_PARAMETER_SET** attribute of the template for the public key.

16603 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
16604 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_VALUE**, and **CKA_PARAMETER_SET**
16605 attributes to the new private key.

16606 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
16607 are not used and must be set to 0.

16608 6.66.7 XMSS^{MT} key pair generation

16609 The XMSS^{MT} key pair generation mechanism, denoted **CKM_XMSSMT_KEY_PAIR_GEN**, is the key pair
16610 generation mechanism for XMSS^{MT}.

16611 The mechanism generates XMSS^{MT} public/private key pairs using an *oid*, as specified in the
16612 **CKA_PARAMETER_SET** attribute of the template for the public key.

16613 All other restrictions detailed in section 6.66.6 apply, using XMSS^{MT} types where necessary.

16614 6.66.8 XMSS and XMSS^{MT} without hashing

16615 The XMSS and XMSS^{MT} without hashing mechanisms, denoted **CKM_XMSS** and **CKM_XMSSMT**
16616 respectively, are mechanisms for single-part signatures and verification.

16617 These mechanisms do not have parameters.

16618 For the purposes of these mechanisms, an XMSS or XMSS^{MT} signature is a byte string with a length
16619 depending on the *oid* provided.

16620 Table 277, XMSS without hashing: Key and Data Length

Function	Key type	Input length	Output length
C_Sign ¹	XMSS Private Key	any	2500-9732 ²
C_Verify ¹	XMSS Public Key	any, 2500-9732 ²	N/A

16621 1 Single-part operations only.
16622 2 Smallest and largest signature sizes from [RFC 8391], including optional parameter sets.

16623 Table 278, XMSS^{MT} without hashing: Key and Data Length

Function	Key type	Input length	Output length
C_Sign ¹	XMSS ^{MT} Private Key	any	4963-104520 ²
C_Verify ¹	XMSS ^{MT} Public Key	any, 4963-104520 ²	N/A

16624 1 Single-part operations only.
16625 2 Smallest and largest signature sizes from [RFC 8391], including optional parameter sets.
16626

16627 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
16628 structure are not used and must be set to 0.

16629 **6.67 ML-DSA**

16630 ML-DSA and HashML-DSA are mechanisms for signatures and verification, following the digital signature
16631 algorithm defined in [FIPS 204].

16632 Table 279, ML-DSA Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ML_DSA_KEY_PAIR_GEN					✓			
CKM_ML_DSA		✓ ¹						
CKM_HASH_ML_DSA		✓ ²						
CKM_HASH_ML_DSA_SHA224		✓						
CKM_HASH_ML_DSA_SHA256		✓						
CKM_HASH_ML_DSA_SHA384		✓						
CKM_HASH_ML_DSA_SHA512		✓						
CKM_HASH_ML_DSA_SHA3_224		✓						
CKM_HASH_ML_DSA_SHA3_256		✓						
CKM_HASH_ML_DSA_SHA3_384		✓						
CKM_HASH_ML_DSA_SHA3_512		✓						
CKM_HASH_ML_DSA_SHAKE128		✓						
CKM_HASH_ML_DSA_SHAKE256		✓						

16633 1 Verification is only for single part verifications or multipart verifications when the **C_VerifySignatureInit** interface is used
16634 2 Single-part operations only

16635

16636 6.67.1 Definitions

16637 This section defines the key type **CKK_ML_DSA** for type **CK_KEY_TYPE** as used in the
16638 **CKA_KEY_TYPE** attribute of all ML-DSA key objects.

16639 Mechanisms:

16640 CKM_ML_DSA_KEY_PAIR_GEN
16641 CKM_ML_DSA
16642 CKM_HASH_ML_DSA
16643 CKM_HASH_ML_DSA_SHA224
16644 CKM_HASH_ML_DSA_SHA256
16645 CKM_HASH_ML_DSA_SHA384
16646 CKM_HASH_ML_DSA_SHA512
16647 CKM_HASH_ML_DSA_SHA3_224
16648 CKM_HASH_ML_DSA_SHA3_256
16649 CKM_HASH_ML_DSA_SHA3_384
16650 CKM_HASH_ML_DSA_SHA3_512
16651 CKM_HASH_ML_DSA_SHAKE128
16652 CKM_HASH_ML_DSA_SHAKE256

16653

16654 **CK_ML_DSA_PARAMETER_SET_TYPE** is used to indicate which ML-DSA parameter set the keys
16655 belong to.

```
16656     typedef CK_ULONG CK_ML_DSA_PARAMETER_SET_TYPE;
```

16657

16658 Parameter set types:

16659 CKP_ML_DSA_44
16660 CKP_ML_DSA_65
16661 CKP_ML_DSA_87

16662

16663 **CK_HEDGE_TYPE** is used to indicate how hedge or deterministic signature variants are handled.

```
16664     typedef CK_ULONG CK_HEDGE_TYPE;
```

16665

16666 Hedge types:

16667 CKH_HEDGE_PREFERRED
16668 CKH_HEDGE_REQUIRED
16669 CKH_DETERMINISTIC_REQUIRED

16670

16671 **CK_SIGN_ADDITIONAL_CONTEXT** is used in the mechanism parameters to supply a NIST defined
16672 context string in signature scheme.

```
16673     typedef struct CK_SIGN_ADDITIONAL_CONTEXT {  
16674         CK_HEDGE_TYPE    hedgeVariant;  
16675         CK_BYTE_PTR       pContext;  
16676         CK_ULONG          ulContextLen;  
16677     } CK_SIGN_ADDITIONAL_CONTEXT;
```

16678

;

16679

16680

16681

16682

16683

16684

16685

16686

16687

16688

16689

CK_HASH_SIGN_ADDITIONAL_CONTEXT is used in the mechanism parameters to supply a NIST defined context string in signature scheme and the hash algorithm.

typedef struct CK_HASH_SIGN_ADDITIONAL_CONTEXT {

CK_HEDGE_TYPE hedgeVariant;

CK_BYTE_PTR pContext;

CK_ULONG ulContextLen;

CK_MECHANISM_TYPE hash;

} CK_HASH_SIGN_ADDITIONAL_CONTEXT;

;

16690

6.67.2 ML-DSA public key objects

16691

16692

16693

16694

16695

ML-DSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_ML_DSA**) hold ML-DSA public keys.

The following table defines the ML-DSA public key object attributes, in addition to the common attributes defined for this object class:

Table 280, ML-DSA Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,3}	CK_ML_DSA_PARAMETER_SET_TYPE	ML-DSA parameter set
CKA_VALUE ^{1,4}	Byte array	Public key as defined in [FIPS 204]

16696

16697

16698

16699

16700

16701

16702

16703

16704

16705

16706

16707

16708

16709

16710

16711

16712

16713

16714

16715

16716

Refer to Table 13 for footnotes

The **CKA_PARAMETER_SET** attribute value selects a predefined set of parameters specified by NIST. The parameter set will select the security level and public key sizes. Tokens may support a subset of the defined parameter sets.

The following is a sample template for creating an ML-DSA public key object:

CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;

CK_KEY_TYPE keyType = CKK_ML_DSA;

CK_UTF8CHAR label[] = "A ML-DSA public key object";

CK_ML_DSA_PARAMETER_SET_TYPE param_set = CKP_ML_DSA_44;

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_PARAMETER_SET, ¶m_set, sizeof(param_set)},

{CKA_VALUE, value, sizeof(value)}

};

6.67.3 ML-DSA private key objects

ML-DSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_ML_DSA**) hold ML-DSA private keys.

The following table defines the ML-DSA private key object attributes, in addition to the common attributes defined for this object class:

Table 281, ML-DSA Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,4,6}	CK_ML_DSA_PARAMETER_SET_TYPE	ML-DSA parameter set
CKA_SEED ^{4,6,7}	Byte array	Seed value (ξ) as defined in ML-DSA.Keygen in [FIPS 204]
CKA_VALUE ^{1,4,6,7}	Byte array	Private key (sk) as defined in ML-DSA.Keygen-internal in [FIPS 204]

Refer to Table 13 for footnotes

At least one of **CKA_SEED** and **CKA_VALUE** must be specified on **C_CreateObject**. Tokens may reject creation requests that only specify one of these values. For highest compatibility applications should set both.

The **CKA_PARAMETER_SET** attribute value selects a predefined set of parameters specified by NIST. The parameter set will select the security level and private key sizes. Tokens may support a subset of the defined parameter sets.

Note that when generating a ML-DSA private key, the parameter set is *not* specified in the key's template. This is because ML-DSA private keys are only generated as part of a ML-DSA key *pair*, and the parameter set for the pair is specified in the template for the ML-DSA public key.

The following is a sample template for creating an ML-DSA private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_ML_DSA;
CK_UTF8CHAR label[] = "A ML-DSA private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_ML_DSA_PARAMETER_SET_TYPE param_set = CKP_ML_DSA_44;
CK_BYTE seed[] = {...};
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_SIGN, &true, sizeof(true)},
    {CKA_PARAMETER_SET, &param_set, sizeof(param_set)},
    {CKA_SEED, seed, sizeof(seed)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.67.4 ML-DSA key pair generation

The ML-DSA key pair generation mechanism, denoted **CKM_ML_DSA_KEY_PAIR_GEN**, is a key pair generation mechanism using ML-DSA.KeyGen() as defined in section 5.1 of [FIPS 204].

It does not have a parameter.

The mechanism generates ML-DSA public/private key pairs with a parameter set, as specified in the **CKA_PARAMETER_SET** attribute of the template for the public key.

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_PARAMETER_SET**, **CKA_SEED** and **CKA_VALUE** attributes to the new private key; other attributes required by the ML-DSA 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 ML-DSA public key in bytes.

6.67.5 ML-DSA Signature

The ML-DSA signature mechanism, denoted **CKM_ML_DSA**, is a mechanism for generating and verifying ML-DSA signatures as defined in sections 5.2 and 5.3 of [FIPS 204], Algorithm 2 ML-DSA.Sign and Algorithm 3 ML-DSA.Verify, using SHAKE256 as hash function. The data passed in is the message M.

It has an optional parameter CK_SIGN_ADDITIONAL_CONTEXT. If no parameter is supplied the hedgeVariant will be **CKH_HEDGE_PREFERRED**, ulContextLen will be zero and pContext will be NULL. On signing, if hedgeVariant is set to **CKH_HEDGE_PREFERRED**, the token may create either a hedged signature or a deterministic signature as specified in [FIPS 204]. If hedgeVariant is set to **CKH_HEDGE_REQUIRED**, the token must produce a hedged signature or fail. If the hedgeVariant is set to **CKH_DETERMINISTIC_REQUIRED**, the token must produce a deterministic signature or fail. On verification the hedgeVariant parameter is ignored.

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 ML-DSA signature.

Table 282, ML-DSA: Key and Data Length

Function	Key type	Input length	Output length
C_Sign	ML-DSA Private Key	any	k
C_Verify ¹	ML-DSA Public Key	any, k	N/A
C_VerifySignature	ML-DSA Public Key	any, k	N/A

¹ Single-part operations only.

For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure specify the supported range of ML-DSA public keys in bytes.

6.67.6 HashML-DSA Signature

The HashML-DSA signature mechanism, denoted **CKM_HASH_ML_DSA** is a single part mechanism for generating and verifying pre-hash ML-DSA signatures as defined in section 5.4 of [FIPS 204], Algorithm 4 HashML-DSA.Sign and Algorithm 5 HashML-DSA.Verify, using the hash function specified in CK_HASH_SIGN_ADDITIONAL_CONTEXT. The data passed in is an already hashed message PH_M.

It has a parameter CK_HASH_SIGN_ADDITIONAL_CONTEXT. On signing, if hedgeVariant is set to **CKH_HEDGE_PREFERRED**, the token may create either a hedged signature or a deterministic signature as specified in [FIPS 204]. If hedgeVariant is set to **CKH_HEDGE_REQUIRED**, the token must produce a hedged signature or fail. If the hedgeVariant is set to **CKH_DETERMINISTIC_REQUIRED**, the token must produce a deterministic signature or fail. On verification the hedgeVariant parameter is ignored.

16798 Constraints on key types and the length of the data are summarized in the following table. In the table, k
16799 is the length in bytes of the ML-DSA signature.

16800 *Table 283, HashML-DSA: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign ¹	ML-DSA Private Key	Length of hash	k
C_Verify ¹	ML-DSA Public Key	any, k	N/A

16801 ¹ Single-part operations only.

16802

16803 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
16804 structure specify the supported range of ML-DSA public keys in bytes.

16805 **6.67.7 HashML-DSA Signature with hashing**

16806 The HashML-DSA with hashing mechanism, denoted **CKM_HASH_ML_DSA_<hash>** where <hash>
16807 identifies a hash function as per Table 284, is a mechanism for single- and multiple-part signatures and
16808 verification for pre-hash ML-DSA signatures as defined in section 5.4 of [FIPS 204], Algorithm 4 HashML-
16809 DSA.Sign and Algorithm 5 HashML-DSA.Verify. This mechanism computes the entire HashML-DSA
16810 specification, including the hashing on token. The data passed in is the message M.

16811

16812 *Table 284, HashML-DSA with hashing: mechanisms and hash functions*

Mechanism	Hash function
CKM_HASH_ML_DSA_SHA224	SHA-224
CKM_HASH_ML_DSA_SHA256	SHA-256
CKM_HASH_ML_DSA_SHA384	SHA-384
CKM_HASH_ML_DSA_SHA512	SHA-512
CKM_HASH_ML_DSA_SHA3_224	SHA3-224
CKM_HASH_ML_DSA_SHA3_256	SHA3-256
CKM_HASH_ML_DSA_SHA3_384	SHA3-384
CKM_HASH_ML_DSA_SHA3_512	SHA3-512
CKM_HASH_ML_DSA_SHAKE128	SHAKE128
CKM_HASH_ML_DSA_SHAKE256	SHAKE256

16813

16814 These mechanisms have an optional parameter CK_SIGN_ADDITIONAL_CONTEXT. If no parameter is
16815 supplied the hedgeVariant will be **CKH_HEDGE_PREFERRED**, ulContextLen will be zero and pContext
16816 will be NULL. On signing, if hedgeVariant is set to **CKH_HEDGE_PREFERRED**, the token may create
16817 either a hedged signature or a deterministic signature as specified in [FIPS 204]. If hedgeVariant is set to
16818 **CKH_HEDGE_REQUIRED**, the token must produce a hedged signature or fail. If the hedgeVariant is set
16819 to **CKH_DETERMINISTIC_REQUIRED**, the token must produce a deterministic signature or fail. On
16820 verification the hedgeVariant parameter is ignored.

16821 Constraints on key types and the length of the data are summarized in the following table. In the table, k
16822 is the length in bytes of the ML-DSA signature.

16823 Table 285, HashML-DSA with hashing: Key and Data Length

Function	Key type	Input length	Output length
C_Sign	ML-DSA Private Key	any	k
C_Verify	ML-DSA Public Key	any, k	N/A
C_VerifySignature	ML-DSA Public Key	any, k	N/A

16824

16825 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**

16826 structure specify the supported range of ML-DSA public keys in bytes.

16827 **6.68 ML-KEM**

16828 ML-KEM is a mechanism for key encapsulation, following the key encapsulation algorithm defined in

16829 [FIPS 203].

16830 Table 286, ML-KEM Mechanisms vs. Functions

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_ML_KEM_KEY_PAIR_GEN					✓			
CKM_ML_KEM								✓

16831

16832 **6.68.1 Definitions**

16833 This section defines the key type **CKK_ML_KEM** for type **CK_KEY_TYPE** as used in the

16834 **CKA_KEY_TYPE** attribute of all ML-KEM key objects.

16835 Mechanisms:

16836 CKM_ML_KEM_KEY_PAIR_GEN

16837 CKM_ML_KEM

16838

16839 **CK_ML_KEM_PARAMETER_SET_TYPE** is used to indicate which ML-KEM parameter set the keys

16840 belong to.

16841 `typedef CK_ULONG CK_ML_KEM_PARAMETER_SET_TYPE;`

16842

16843 Parameter set types:

16844 CKP_ML_KEM_512

16845 CKP_ML_KEM_768

16846 CKP_ML_KEM_1024

16847 **6.68.2 ML-KEM public key objects**

16848 ML-KEM public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_ML_KEM**) hold ML-KEM

16849 public keys.

16850 The following table defines the ML-KEM public key object attributes, in addition to the common attributes

16851 defined for this object class:

16852 Table 287, ML-KEM Public Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,3}	CK_ML_KEM_PARAMETER_SET_TYPE	ML-KEM parameter set
CKA_VALUE ^{1,4}	Byte array	Public value i.e. encapsulation key ek as defined in [FIPS 203]

16853 Refer to Table 13 for footnotes

16854
16855 The **CKA_PARAMETER_SET** attribute value selects a predefined set of parameters specified by NIST.
16856 The parameter set will select the security level and public key sizes. Tokens may support a subset of the
16857 defined parameter sets.

16858
16859 The following is a sample template for creating an ML-KEM public key object:

```
16860 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
16861 CK_KEY_TYPE keyType = CKK_ML_KEM;  
16862 CK_UTF8CHAR label[] = "A ML-KEM public key object";  
16863 CK_ML_KEM_PARAMETER_SET_TYPE param_set = CKP_ML_KEM_512;  
16864 CK_BYTE value[] = {...};  
16865 CK_BBOOL true = CK_TRUE;  
16866 CK_ATTRIBUTE template[] = {  
16867     {CKA_CLASS, &class, sizeof(class)},  
16868     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
16869     {CKA_TOKEN, &true, sizeof(true)},  
16870     {CKA_LABEL, label, sizeof(label)-1},  
16871     {CKA_ENCAPSULATE, &true, sizeof(true)},  
16872     {CKA_PARAMETER_SET, &param_set, sizeof(param_set)},  
16873     {CKA_VALUE, value, sizeof(value)}  
16874 };
```

16875 **6.68.3 ML-KEM private key objects**

16876 ML-KEM private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_ML_KEM**) hold ML-KEM
16877 private keys.

16878 The following table defines the ML-KEM private key object attributes, in addition to the common attributes
16879 defined for this object class:

16880 Table 288, ML-KEM Private Key Object Attributes

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,4,6}	CK_ML_KEM_PARAMETER_SET_TYPE	ML-KEM parameter set
CKA_SEED ^{4,6,7}	Byte array	Randomness value (d z) as defined in ML-KEM.Keygen in [FIPS 203]
CKA_VALUE ^{1,4,6,7}	Byte array	Private value i.e. decapsulation key dk as defined in [FIPS 203]

16881 Refer to Table 13 for footnotes

16882
16883 At least one of **CKA_SEED** and **CKA_VALUE** must be specified on **C_CreateObject**. Tokens may reject
16884 creation requests that only specify one of these values. For highest compatibility applications should set
16885 both.

The **CKA_PARAMETER_SET** attribute value selects a predefined set of parameters specified by NIST. The parameter set will select the security level and private key sizes. Tokens may support a subset of the defined parameter sets.

Note that when generating a ML-KEM private key, the parameter set is *not* specified in the key's template. This is because ML-KEM private keys are only generated as part of a ML-KEM key *pair*, and the parameter set for the pair is specified in the template for the ML-KEM public key.

The following is a sample template for creating an ML-KEM private key object:

```
CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;
CK_KEY_TYPE keyType = CKK_ML_KEM;
CK_UTF8CHAR label[] = "A ML-KEM private key object";
CK_BYTE subject[] = {...};
CK_BYTE id[] = {123};
CK_ML_KEM_PARAMETER_SET_TYPE param_set = CKP_ML_KEM_512;
CK_BYTE seed[] = {...};
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_DECAPSULATE, &true, sizeof(true)},
    {CKA_PARAMETER_SET, &param_set, sizeof(param_set)},
    {CKA_SEED, seed, sizeof(seed)},
    {CKA_VALUE, value, sizeof(value)}
};
```

6.68.4 ML-KEM key pair generation

The ML-KEM key pair generation mechanism, denoted **CKM_ML_KEM_KEY_PAIR_GEN**, is a key pair generation mechanism using Algorithm 19 ML-KEM.KeyGen as defined in [FIPS 203].

It does not have a parameter.

The mechanism generates ML-KEM public/private key pairs with a parameter set, as specified in the **CKA_PARAMETER_SET** attribute of the template for the public key.

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_PARAMETER_SET**, **CKA_SEED** and **CKA_VALUE** attributes to the new private key; other attributes required by the ML-KEM 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 ML-KEM public key in bytes.

6.68.5 ML-KEM Key Agreement

The ML-KEM Key Agreement mechanism, denoted **CKM_ML_KEM**, is a mechanism for key encapsulation and decapsulation using Algorithm 20 ML-KEM.Encaps and Algorithm 21 ML-KEM.Decaps respectively. Both are defined in [FIPS 203].

16932 It has no parameters.

16933 When used in **C_EncapsulateKey**, this mechanism generates a secret key and an encapsulated

16934 ciphertext from a ML-KEM public key using ML-KEM.Encaps.

16935 When used in **C_DecapsulateKey**, this mechanism generates a secret key from an encapsulated

16936 ciphertext and a ML-KEM private key using ML-KEM.Decaps.

16937 The mechanism contributes the result as the **CKA_VALUE** attribute of the new key; other attributes

16938 required by the key type must be specified in the template.

16939 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**

16940 structure specify the supported range of ML-KEM public keys in bytes.

16941 **6.69 SLH-DSA**

16942 SLH-DSA and HashSLH-DSA are mechanisms for signatures and verification, following the digital

16943 signature algorithm defined in [FIPS 205].

16944 *Table 289, SLH-DSA Mechanisms vs. Functions*

Mechanism	Functions							
	Encrypt & Decrypt	Sign & Verify	Sign Recover & Verify Recover	Digest	Gen. Key/ Key Pair	Wrap & Unwrap	Derive	Encap sulate & Decap sulate
CKM_SLH_DSA_KEY_PAIR_GEN					✓			
CKM_SLH_DSA		✓ ¹						
CKM_HASH_SLH_DSA		✓ ²						
CKM_HASH_SLH_DSA_SHA224		✓						
CKM_HASH_SLH_DSA_SHA256		✓						
CKM_HASH_SLH_DSA_SHA384		✓						
CKM_HASH_SLH_DSA_SHA512		✓						
CKM_HASH_SLH_DSA_SHA3_224		✓						
CKM_HASH_SLH_DSA_SHA3_256		✓						
CKM_HASH_SLH_DSA_SHA3_384		✓						
CKM_HASH_SLH_DSA_SHA3_512		✓						
CKM_HASH_SLH_DSA_SHAKE128		✓						
CKM_HASH_SLH_DSA_SHAKE256		✓						

16945 1.Verification is only for single part verifications or multipart verifications when the **C_VerifySignatureInit** interface is used

16946 2 Single-part operations only.

16947

16948 **6.69.1 Definitions**

16949 This section defines the key type **CKK_SLH_DSA** for type **CK_KEY_TYPE** as used in the

16950 **CKA_KEY_TYPE** attribute of all SLH-DSA key objects.

16951 Mechanisms:

16952 CKM_SLH_DSA_KEY_PAIR_GEN

16953 CKM_SLH_DSA
16954 CKM_HASH_SLH_DSA
16955 CKM_HASH_SLH_DSA_SHA224
16956 CKM_HASH_SLH_DSA_SHA256
16957 CKM_HASH_SLH_DSA_SHA384
16958 CKM_HASH_SLH_DSA_SHA512
16959 CKM_HASH_SLH_DSA_SHA3_224
16960 CKM_HASH_SLH_DSA_SHA3_256
16961 CKM_HASH_SLH_DSA_SHA3_384
16962 CKM_HASH_SLH_DSA_SHA3_512
16963 CKM_HASH_SLH_DSA_SHAKE128
16964 CKM_HASH_SLH_DSA_SHAKE256

16965

16966 **CK_SLH_DSA_PARAMETER_SET_TYPE** is used to indicate which SLH-DSA parameter set the keys
16967 belong to.

16968 typedef CK_ULONG CK_SLH_DSA_PARAMETER_SET_TYPE;
16969

16970 Parameter set types:

16971 CKP_SLH_DSA_SHA2_128S
16972 CKP_SLH_DSA_SHAKE_128S
16973 CKP_SLH_DSA_SHA2_128F
16974 CKP_SLH_DSA_SHAKE_128F
16975 CKP_SLH_DSA_SHA2_192S
16976 CKP_SLH_DSA_SHAKE_192S
16977 CKP_SLH_DSA_SHA2_192F
16978 CKP_SLH_DSA_SHAKE_192F
16979 CKP_SLH_DSA_SHA2_256S
16980 CKP_SLH_DSA_SHAKE_256S
16981 CKP_SLH_DSA_SHA2_256F
16982 CKP_SLH_DSA_SHAKE_256S

16983 **6.69.2 SLH-DSA public key objects**

16984 SLH-DSA public key objects (object class **CKO_PUBLIC_KEY**, key type **CKK_SLH_DSA**) hold SLH-DSA
16985 public keys.

16986 The following table defines the SLH-DSA public key object attributes, in addition to the common attributes
16987 defined for this object class:

16988 *Table 290, SLH-DSA Public Key Object Attributes*

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,3}	CK_SLH_DSA_PARAMETER_SET_TYPE	SLH-DSA parameter set
CKA_VALUE ^{1,4}	Byte array	Public key as defined in [FIPS 205]

16989 ¹ Refer to Table 13 for footnotes

16990

16991 The **CKA_PARAMETER_SET** attribute value selects a predefined set of parameters specified by NIST.
16992 The parameter set will select the security level and public key sizes. Tokens may support a subset of the
16993 defined parameter sets.

16994

16995 The following is a sample template for creating an SLH-DSA public key object:

```
16996 CK_OBJECT_CLASS class = CKO_PUBLIC_KEY;  
16997 CK_KEY_TYPE keyType = CKK_SLH_DSA;  
16998 CK_UTF8CHAR label[] = "A SLH-DSA public key object";  
16999 CK_SLH_DSA_PARAMETER_SET_TYPE param_set =  
17000     CKP_SLH_DSA_SHAKE_128S;  
17001 CK_BYTE value[] = {...};  
17002 CK_BBOOL true = CK_TRUE;  
17003 CK_ATTRIBUTE template[] = {  
17004     {CKA_CLASS, &class, sizeof(class)},  
17005     {CKA_KEY_TYPE, &keyType, sizeof(keyType)},  
17006     {CKA_TOKEN, &true, sizeof(true)},  
17007     {CKA_LABEL, label, sizeof(label)-1},  
17008     {CKA_PARAMETER_SET, &param_set, sizeof(param_set)},  
17009     {CKA_VALUE, value, sizeof(value)}  
17010 };
```

17011 **6.69.3 SLH-DSA private key objects**

17012 SLH-DSA private key objects (object class **CKO_PRIVATE_KEY**, key type **CKK_SLH_DSA**) hold SLH-
17013 DSA private keys.

17014 The following table defines the SLH-DSA private key object attributes, in addition to the common
17015 attributes defined for this object class:

17016 *Table 291, SLH-DSA Private Key Object Attributes*

Attribute	Data Type	Meaning
CKA_PARAMETER_SET ^{1,4,6}	CK_SLH_DSA_PARAMETER_SET_TYPE	SLH-DSA parameter set
CKA_VALUE ^{1,4,6,7}	Byte array	Private key as defined in [FIPS 205]

17017 ¹Refer to Table 13 for footnotes

17018

17019 The **CKA_PARAMETER_SET** attribute value selects a predefined set of parameters specified by NIST.
17020 The parameter set will select the security level and private key sizes. Tokens may support a subset of the
17021 defined parameter sets.

17022 Note that when generating a SLH-DSA private key, the parameter set is *not* specified in the key's
17023 template. This is because SLH-DSA private keys are only generated as part of a SLH-DSA key *pair*, and
17024 the parameter set for the pair is specified in the template for the SLH-DSA public key.

17025

17026 The following is a sample template for creating an SLH-DSA private key object:

```
17027 CK_OBJECT_CLASS class = CKO_PRIVATE_KEY;  
17028 CK_KEY_TYPE keyType = CKK_SLH_DSA;  
17029 CK_UTF8CHAR label[] = "A SLH-DSA private key object";  
17030 CK_BYTE subject[] = {...};  
17031 CK_BYTE id[] = {123};  
17032 CK_SLH_DSA_PARAMETER_SET_TYPE param_set =
```

```

17033         CKP_SLH_DSA_SHAKE_128S;
17034     CK_BYTE value[] = {...};
17035     CK_BBOOL true = CK_TRUE;
17036     CK_ATTRIBUTE template[] = {
17037         {CKA_CLASS, &class, sizeof(class)},
17038         {CKA_KEY_TYPE, &keyType, sizeof(keyType)},
17039         {CKA_TOKEN, &true, sizeof(true)},
17040         {CKA_LABEL, label, sizeof(label)-1},
17041         {CKA_SUBJECT, subject, sizeof(subject)},
17042         {CKA_ID, id, sizeof(id)},
17043         {CKA_SENSITIVE, &true, sizeof(true)},
17044         {CKA_SIGN, &true, sizeof(true)},
17045         {CKA_PARAMETER_SET, &param_set, sizeof(param_set)},
17046         {CKA_VALUE, value, sizeof(value)}
17047     };

```

17048 6.69.4 SLH-DSA key pair generation

17049 The SLH-DSA key pair generation mechanism, denoted **CKM_SLH_DSA_KEY_PAIR_GEN**, is a key pair
17050 generation mechanism as defined in Algorithm 21 of [FIPS 205].

17051 It does not have a parameter.

17052 The mechanism generates SLH-DSA public/private key pairs with a parameter set, as specified in the
17053 **CKA_PARAMETER_SET** attribute of the template for the public key.

17054 The mechanism contributes the **CKA_CLASS**, **CKA_KEY_TYPE**, and **CKA_VALUE** attributes to the new
17055 public key and the **CKA_CLASS**, **CKA_KEY_TYPE**, **CKA_PARAMETER_SET**, and **CKA_VALUE**
17056 attributes to the new private key; other attributes required by the SLH-DSA public and private key types
17057 must be specified in the templates.

17058 For this mechanism, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO** structure
17059 specify the supported range of SLH-DSA public key in bytes.

17060 6.69.5 SLH-DSA Signature

17061 The SLH-DSA signature mechanism, denoted **CKM_SLH_DSA**, is a mechanism for generating and
17062 verifying SLH-DSA signatures as defined in Algorithm 22 *slh_sign* and Algorithm 24 *slh_verify* of [FIPS
17063 205], using the hash function determined from **CK_SLH_DSA_PARAMETER_SET_TYPE** and section 11
17064 of [FIPS 205]. The data passed in is the message M. Verification is only for single part verifications or
17065 multipart verifications when the **C_VerifySignatureInit** interface is used.

17066 It has an optional parameter **CK_SIGN_ADDITIONAL_CONTEXT** (see section 6.67.1). If no parameter is
17067 supplied *ulContextLen* will be zero and *pContext* will be NULL. On signing, if *hedgeVariant* is set to
17068 **CKH_HEDGE_PREFERRED**, the token may create either a hedged signature or a deterministic signature
17069 as specified in [FIPS 205]. If *hedgeVariant* is set to **CKH_HEDGE_REQUIRED**, the token must produce a
17070 hedged signature or fail. If the *hedgeVariant* is set to **CKH_DETERMINISTIC_REQUIRED**, the token
17071 must produce a deterministic signature or fail. On verification the *hedgeVariant* parameter is ignored.

17072 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*
17073 is the length in bytes of the SLH-DSA signature.

17074 Table 292, SLH-DSA: Key and Data Length

Function	Key type	Input length	Output length
C_Sign	SLH-DSA Private Key	any	k
C_Verify ¹	SLH-DSA Public Key	any, k	N/A
C_VerifySignature	SLH-DSA Public Key	any, k	N/A

17075 ¹ Single-part operations only.

17076
17077 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
17078 structure specify the supported range of SLH-DSA public keys in bytes.

17079 **6.69.6 HashSLH-DSA Signature**

17080 The HashSLH-DSA signature mechanism, denoted **CKM_HASH_SLH_DSA**, is a single part mechanism
17081 for generating and verifying pre-hash SLH-DSA signatures defined in Algorithm 23 hash_slh_sign of
17082 [FIPS 205], using the hash function specified in CK_HASH_SIGN_ADDITIONAL_CONTEXT. The data
17083 passed in is an already hashed message PH_M.

17084 It has a parameter CK_HASH_SIGN_ADDITIONAL_CONTEXT (see section 6.67.1). On signing, if
17085 hedgeVariant is set to **CKH_HEDGE_PREFERRED**, the token may create either a hedged signature or a
17086 deterministic signature as specified in [FIPS 205]. If hedgeVariant is set to **CKH_HEDGE_REQUIRED**,
17087 the token must produce a hedged signature or fail. If the hedgeVariant is set to
17088 **CKH_DETERMINISTIC_REQUIRED**, the token must produce a deterministic signature or fail. On
17089 verification the hedgeVariant parameter is ignored.

17090 Constraints on key types and the length of the data are summarized in the following table. In the table, *k*
17091 is the length in bytes of the SLH-DSA signature.

17092 Table 293, HashSLH-DSA: Key and Data Length

Function	Key type	Input length	Output length
C_Sign ¹	SLH-DSA Private Key	Length of hash	k
C_Verify ¹	SLH-DSA Public Key	any, k	N/A

17093 ¹ Single-part operations only.

17094
17095 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
17096 structure specify the supported range of SLH-DSA public keys in bytes.

17097 **6.69.7 HashSLH-DSA Signature with hashing**

17098 The HashSLH-DSA with hashing mechanism, denoted **CKM_HASH_SLH_DSA_<hash>** where <hash>
17099 identifies a hash function as per Table 294, is a mechanism for single- and multiple-part signatures and
17100 verification for HashSLH-DSA. This mechanism computes the entire HashSLH-DSA specification,
17101 including the hashing on token. The data passed in is the message M.

17102
17103 Table 294, HashSLH-DSA with hashing: mechanisms and hash functions

Mechanism	Hash function
CKM_HASH_SLH_DSA_SHA224	SHA-224
CKM_HASH_SLH_DSA_SHA256	SHA-256
CKM_HASH_SLH_DSA_SHA384	SHA-384
CKM_HASH_SLH_DSA_SHA512	SHA-512

CKM_HASH_SLH_DSA_SHA3_224	SHA3-224
CKM_HASH_SLH_DSA_SHA3_256	SHA3-256
CKM_HASH_SLH_DSA_SHA3_384	SHA3-384
CKM_HASH_SLH_DSA_SHA3_512	SHA3-512
CKM_HASH_SLH_DSA_SHAKE128	SHAKE128
CKM_HASH_SLH_DSA_SHAKE256	SHAKE256

17104

17105 These mechanisms have an optional parameter CK_SIGN_ADDITIONAL_CONTEXT. If no parameter is
 17106 supplied the hedgeVariant will be **CKH_HEDGE_PREFERRED**, ulContextLen will be zero and pContext
 17107 will be NULL. On signing, if hedgeVariant is set to **CKH_HEDGE_PREFERRED**, the token may create
 17108 either a hedged signature or a deterministic signature as specified in [FIPS 205]. If hedgeVariant is set to
 17109 **CKH_HEDGE_REQUIRED**, the token must produce a hedged signature or fail. If the hedgeVariant is set
 17110 to **CKH_DETERMINISTIC_REQUIRED**, the token must produce a deterministic signature or fail. On
 17111 verification the hedgeVariant parameter is ignored.

17112 Constraints on key types and the length of the data are summarized in the following table. In the table, k
 17113 is the length in bytes of the SLH-DSA signature.

17114 *Table 295, Hash SLH-DSA with hashing: Key and Data Length*

Function	Key type	Input length	Output length
C_Sign	SLH-DSA Private Key	any	k
C_Verify	SLH-DSA Public Key	any, k	N/A
C_VerifySignature	SLH-DSA Public Key	any, k	N/A

17115

17116 For these mechanisms, the *ulMinKeySize* and *ulMaxKeySize* fields of the **CK_MECHANISM_INFO**
 17117 structure specify the supported range of SLH-DSA public keys in bytes.

17118 7 PKCS #11 Implementation Conformance

17119 7.1 PKCS #11 Consumer Implementation Conformance

17120 An implementation is a conforming PKCS #11 Consumer if the implementation meets the conditions
17121 specified in one or more consumer profiles specified in **[PKCS11-Prof]**.

17122 A PKCS #11 consumer implementation SHALL be a conforming PKCS #11 Consumer.

17123 If a PKCS #11 consumer implementation claims support for a particular consumer profile, then the
17124 implementation SHALL conform to all normative statements within the clauses specified for that profile
17125 and for any subclauses to each of those clauses.

17126 7.2 PKCS #11 Provider Implementation Conformance

17127 An implementation is a conforming PKCS #11 Provider if the implementation meets the conditions
17128 specified in one or more provider profiles specified in **[PKCS11-Prof]**.

17129 A PKCS #11 provider implementation SHALL be a conforming PKCS #11 Provider.

Appendix A. References

This appendix contains the normative and informative references that are used in this document. While any hyperlinks included in this appendix were valid at the time of publication, OASIS cannot guarantee their long-term validity.

A.1 Normative References

The following documents are referenced in such a way that some or all of their content constitutes requirements of this document.

[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>

[CHACHA]

D. Bernstein, "ChaCha, a variant of Salsa20", January 2008.
URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>

[DH]

W. Diffie, M. Hellman. "New Directions in Cryptography", November 1976.
URL: <http://www-ee.stanford.edu/~hellman/publications/24.pdf>

[FIPS 203]

NIST. "FIPS 203: Module-Lattice-Based Key-Encapsulation Mechanism Standard", August 2024.
URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.203.pdf>

[FIPS 204]

NIST. "FIPS 204: Module-Lattice-Based Digital Signature Standard", August 2024.
URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.204.pdf>

[FIPS 205]

NIST. "FIPS 205: Stateless Hash-Based Digital Signature Standard", August 2024.
URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.205.pdf>

[FIPS PUB 46-3]

NIST. "FIPS PUB 46-3: Data Encryption Standard", October 1999.
URL: <http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf>

[FIPS PUB 81]

NIST. "FIPS PUB 81: DES Modes of Operation", December 1980.
URL: <http://csrc.nist.gov/publications/fips/fips81/fips81.htm>

17173 **[FIPS PUB 180-4]**
 17174 NIST. "FIPS PUB 180-4: "Secure Hash Standard (SHS)", Augut 2015.
 17175 URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>

17176 **[FIPS PUB 186-4]**
 17177 NIST. "FIPS PUB 186-4: Digital Signature Standard", July 2013.
 17178 URL: <http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>

17179 **[FIPS PUB 202]**
 17180 NIST. "FIPS PUB 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions",
 17181 August 2015.
 17182 URL: <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf>

17183 **[MD2]**
 17184 B. Kaliski. RSA Laboratories. "The MD2 Message-Digest Algorithm", April 1992.
 17185 URL: <https://www.ietf.org/rfc/rfc1319.txt>

17186 **[MD5]**
 17187 RSA Data Security. R. Rivest. "The MD5 Message-Digest Algorithm", April 1992.
 17188 URL: <https://www.ietf.org/rfc/rfc1321.txt>

17189 **[NIST SP800-38B]**
 17190 NIST. "Recommendation for Block Cipher Modes of Operation: the CMAC Mode for Authentication" May
 17191 2005.
 17192 URL: <https://csrc.nist.gov/pubs/sp/800/38/b/upd1/final>

17193 **[NIST SP800-56A]**
 17194 NIST. "Special Publication 800-56A Revision 2: Recommendation for Pair-Wise Key Establishment
 17195 Schemes Using Discrete Logarithm Cryptography" May 2013.
 17196 URL: <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>

17197 **[NIST SP 800-56B]**
 17198 NIST. "Special Publication 800-56B Revision 2: Recommendation for Pair-Wise Key-Establishment Using
 17199 Integer Factorization Cryptography" March 2019.
 17200 URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Br2.pdf>

17201 **[NIST SP800-108]**
 17202 NIST. "Special Publication 800-108 (Revised): Recommendation for Key Derivation Using Pseudorandom
 17203 Functions", October 2009.
 17204 URL: <https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-108.pdf>

17205 **[NIST SP800-208]**
 17206 NIST "Special Publication 800-208: Recommendation for Stateful Hash-Based Signature Schemes",
 17207 October 2020.
 17208 URL: <https://csrc.nist.gov/publications/detail/sp/800/208/final>

17209 **[OAEP]**
 17210 M. Bellare, P. Rogaway. "Optimal Asymmetric Encryption – How to Encrypt with RSA", November 1995.

17211 **[PKCS11-Hist]**
 17212 PKCS #11 Cryptographic Token Interface Historical Mechanisms Specification Version 3.0. Edited by
 17213 Chris Zimman and Dieter Bong. Latest stage. <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>
 17214 <https://docs.oasis-open.org/pkcs11/pkcs11-hist/v3.0/pkcs11-hist-v3.0.html>

17215 **[PKCS11-Prof]**
 17216 PKCS #11 Profiles Version 3.2. Edited by Tim Hudson. Latest stage: [https://docs.oasis-](https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.2/pkcs11-profiles-v3.2.html)
 17217 [open.org/pkcs11/pkcs11-profiles/v3.2/pkcs11-profiles-v3.2.html](https://docs.oasis-open.org/pkcs11/pkcs11-profiles/v3.2/pkcs11-profiles-v3.2.html).

17218 **[PKCS11-UG]**
 17219 PKCS #11 Usage Guide Version 3.2. Edited by Dieter Bong. Latest stage: [https://docs.oasis-](https://docs.oasis-open.org/pkcs11/pkcs11-ug/v3.2/pkcs11-ug-v3.2.html)
 17220 [open.org/pkcs11/pkcs11-ug/v3.2/pkcs11-ug-v3.2.html](https://docs.oasis-open.org/pkcs11/pkcs11-ug/v3.2/pkcs11-ug-v3.2.html)

17221 **[PKCS #1] [RFC 8017]**
17222 K. Moriarty, B. Kaliski, J. Jonsson, A. Rusch. RFC 8017 “PKCS #1: RSA Cryptography
17223 Specifications Version 2.2”, November 2016
17224 URL: <https://www.rfc-editor.org/rfc/pdf/rfc8017.txt.pdf>

17225 **[PKCS #3]**
17226 RSA Laboratories. “Diffie-Hellman Key-Agreement Standard. v1.4”, November 1993.
17227 URL: https://www.teletrust.de/fileadmin/files/oid/oid_pkcs-3v1-4.pdf

17228 **[PKCS #5]**
17229 K. Moriarty, B. Kaliski, A. Rusch. RFC 8018. “PKCS #5: Password-Based Cryptography Specification
17230 Version 2.1”, January 2017
17231 URL: <https://www.rfc-editor.org/rfc/pdf/rfc8018.txt.pdf>

17232 **[PKCS #7]**
17233 B.Kaliski. “PKCS #7 Cryptographic Message Syntax Version 1.5”, March 1998
17234 URL: <https://www.rfc-editor.org/rfc/pdf/rfc2315.txt.pdf>

17235 **[PKCS #8]**
17236 B. Kaliski. RFC 5208 “Public-Key Cryptography Standards (PKCS) #8: Private-Key Information Syntax
17237 Specification Version 1.2”, May 2008, obsoleted by RFC 5258 S.Turner “Asymmetric Key Packages”,
17238 August 2010
17239 URL: <https://www.rfc-editor.org/rfc/pdf/rfc5958.txt.pdf>

17240 **[PKCS #12]**
17241 K. Moriarty, M. Nystrom, S. Parkinson, A. Rusch, M. Scott. “PKCS #12 Personal Information Exchange
17242 Syntax v1.1”, July 2014.
17243 URL: <https://www.rfc-editor.org/rfc/pdf/rfc7292.txt.pdf>

17244 **[POLY1305]**
17245 D.J. Bernstein. “The Poly1305-AES message-authentication code”, January 2005.
17246 URL: <https://cr.yp.to/mac/poly1305-20050329.pdf>

17247 **[RFC 2119]** Bradner, S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC
17248 2119, DOI 10.17487/RFC 2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.

17249 **[RFC 2279]**
17250 F. Yergeau. RFC 2279: “UTF-8, a transformation format of ISO 10646 Alis Technologies”, January 1998.
17251 URL: <http://www.ietf.org/rfc/rfc2279.txt>

17252 **[RFC 2409]**
17253 D. Harkins, D.Carrel. RFC 2409: “The Internet Key Exchange (IKE)”, November 1998.
17254 URL: <https://tools.ietf.org/html/rfc2409>

17255 **[RFC 2534]**
17256 Masinter, L., Wing, D., Mutz, A., and K. Holtman. RFC 2534: “Media Features for Display, Print, and Fax”.
17257 March 1999.
17258 URL: <http://www.ietf.org/rfc/rfc2534.txt>

17259 **[RFC 5652]**
17260 R. Housley. RFC 5652: “Cryptographic Message Syntax”, September 2009.
17261 URL: <http://www.ietf.org/rfc/rfc5652.txt>

17262 **[RFC 5996]**
17263 C. Kaufman, P. Hoffman, Y. Nir, P. Eronen. RFC 5996: “Internet Key Exchange Protocol Version 2
17264 (IKEv2)”, September 2010.
17265 URL: <https://tools.ietf.org/html/rfc5996>

17266
17267

17268 **[RFC 8174]**
 17269 Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI
 17270 10.17487/RFC8174, May 2017.
 17271 URL: <https://www.rfc-editor.org/info/rfc8174>
 17272 **[RFC 8391]**
 17273 A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, A. Mohaisen. RFC 8391: "XMSS: eXtended Merkle
 17274 Signature Scheme", May 2018.
 17275 URL: <https://tools.ietf.org/html/rfc8391>
 17276 **[RFC 8554]**
 17277 D. McGrew, m. Curcio, S. Fluhrer. RFC 8554: "Leighton-Micali Hash-Based Signatures", April 2019.
 17278 URL: <https://tools.ietf.org/html/rfc8554>
 17279 **[RIPEMD]**
 17280 H. Dobbertin, A. Bosselaers, B. Preneel. "The hash function RIPEMD-160", February 2012.
 17281 URL: <http://homes.esat.kuleuven.be/~bosselae/ripemd160.html>
 17282 **[SALSA]**
 17283 D. Bernstein, "ChaCha, a variant of Salsa20", January 2008.
 17284 URL: <http://cr.yp.to/chacha/chacha-20080128.pdf>
 17285 **[TLS]** [RFC 2246]
 17286 Dierks, T. and C. Allen, "The TLS Protocol Version 1.0", RFC 2246, January 1999. URL:
 17287 <http://www.ietf.org/rfc/rfc2246.txt> , superseded by [RFC4346] Dierks, T. and E. Rescorla, "The Transport
 17288 Layer Security (TLS) Protocol Version 1.1", RFC 4346, April 2006.
 17289 URL: <http://www.ietf.org/rfc/rfc4346.txt> , which was superseded by [TLS12].
 17290 **[TLS12]** [RFC5246]
 17291 Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August
 17292 2008.
 17293 URL: <http://www.ietf.org/rfc/rfc5246.txt>
 17294 **[TWOFISH]**
 17295 B. Schneier, J. Kelsey, D. Whiting, C. Hall, N. Ferguson. "Twofish: A 128-Bit Block Cipher", June 1998.
 17296 URL: <https://www.schneier.com/academic/twofish/>
 17297 **[X.680]**
 17298 ITU-T. Information Technology — Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.
 17299 July 2002. Identical to ISO/IEC 8824-1
 17300 **[X.690]**
 17301 ITU-T. Information Technology — ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER),
 17302 Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). July 2002. Identical to
 17303 ISO/IEC 8825-1
 17304

17305 **A.2 Informative References**

17306 The following referenced documents are not required for the application of this document but may assist
 17307 the reader with regard to a particular subject area.
 17308
 17309 **[AES KEYWRAP]**
 17310 National Institute of Standards and Technology, NIST Special Publication 800-38F, Recommendation for
 17311 Block Cipher Modes of Operation: Methods for Key Wrapping, December 2012,
 17312 <http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38F.pdf>
 17313 **[ANSI C]**
 17314 ANSI/ISO. American National Standard for Programming Languages – C. 1990.

17315 **[ANSI X9.31]**
 17316 Accredited Standards Committee X9. Digital Signatures Using Reversible Public Key Cryptography for the
 17317 Financial Services Industry (rDSA). 1998.

17318 **[ANSI X9.42]**
 17319 Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry:
 17320 Agreement of Symmetric Keys Using Discrete Logarithm Cryptography. 2003.

17321 **[ANSI X9.62]**
 17322 Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: The
 17323 Elliptic Curve Digital Signature Algorithm (ECDSA). November 2005
 17324 URL: [ANSI X9.62 - Public Key Cryptography for the Financial Services Industry: the Elliptic Curve Digital](#)
 17325 [Signature Algorithm \(ECDSA\) | GlobalSpec](#)

17326 **[ANSI X9.63]**
 17327 Accredited Standards Committee X9. Public Key Cryptography for the Financial Services Industry: Key
 17328 Agreement and Key Transport Using Elliptic Curve Cryptography. 2001.
 17329 URL: <http://webstore.ansi.org/RecordDetail.aspx?sku=X9.63-2011>

17330 **[BRAINPOOL]**
 17331 ECC Brainpool Standard Curves and Curve Generation, v1.0, 19.10.2005
 17332 URL: <http://www.ecc-brainpool.org>

17333 **[CC/PP]**
 17334 W3C. Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies. World Wide Web
 17335 Consortium, January 2004.
 17336 URL: <http://www.w3.org/TR/CCPP-struct-vocab/>

17337 **[CT-KIP]**
 17338 RSA Laboratories. Cryptographic Token Key Initialization Protocol. Version 1.0, December 2005.

17339 **[ISO/IEC 8824-1]**
 17340 ISO. Information Technology-- Abstract Syntax Notation One (ASN.1): Specification of Basic Notation.
 17341 2002.

17342 **[ISO/IEC 8825-1]**
 17343 ISO. Information Technology—ASN.1 Encoding Rules: Specification of Basic Encoding Rules (BER),
 17344 Canonical Encoding Rules (CER), and Distinguished Encoding Rules (DER). 2002.

17345 **[ISO/IEC 9594-1]**
 17346 ISO. Information Technology — Open Systems Interconnection — The Directory: Overview of Concepts,
 17347 Models and Services. 2001.

17348 **[ISO/IEC 9594-8]**
 17349 ISO. Information Technology — Open Systems Interconnection — The Directory: Public-key and Attribute
 17350 Certificate Frameworks. 2001

17351 **[ISO/IEC 9796]**
 17352 ISO. Information Technology — Security Techniques — Digital Signature Scheme Giving Message
 17353 Recovery — Part 2: Integer factorization based mechanisms. 2002.

17354 **[Java MIDP]**
 17355 Java Community Process. Mobile Information Device Profile for Java 2 Micro Edition. November 2002.
 17356 URL: <http://jcp.org/jsr/detail/118.jsp>

17357 **[LEGIFRANCE]**
 17358 Avis relatif aux paramètres de courbes elliptiques définis par l'Etat français (Publication of Elliptic Curve
 17359 parameters by the French state)
 17360 URL: <https://www.legifrance.gouv.fr/affichTexte.do?cidTexte=JORFTEXT000024668816>

17361

17362 **[MeT-PTD]**
 17363 MeT. MeT PTD Definition – Personal Trusted Device Definition, Version 1.0, February 2003.
 17364 URL: <http://www.mobiletransaction.org>

17365 **[NIST AES CTS]**
 17366 National Institute of Standards and Technology, Addendum to NIST Special Publication 800-38A,
 17367 “Recommendation for Block Cipher Modes of Operation: Three Variants of Ciphertext Stealing for CBC
 17368 Mode”
 17369 URL: http://csrc.nist.gov/publications/nistpubs/800-38a/addendum-to-nist_sp800-38A.pdf

17370 **[PCMCIA]**
 17371 Personal Computer Memory Card International Association. *PC Card Standard*, Release 2.1, July 1993.

17372 **[RFC 1421]**
 17373 J. Linn, “Privacy Enhancement for Internet Electronic Mail”, IETF RFC 1421, February 1993.
 17374 URL: <http://www.ietf.org/rfc/rfc1421.txt>.

17375 **[RFC 2104]**
 17376 H. Krawczyk et al, “HMAC: Keyed-Hashing for Message Authentication”, IETF RFC 2104, February 1997.
 17377 URL: <http://www.ietf.org/rfc/rfc2104.txt>.

17378 **[RFC 2865]**
 17379 Rigney et al, “Remote Authentication Dial In User Service (RADIUS)”, IETF RFC 2865, June 2000.
 17380 URL: <http://www.ietf.org/rfc/rfc2865.txt>.

17381 **[RFC 3552]**
 17382 Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC
 17383 3552, DOI 10.17487/RFC3552, July 2003.
 17384 URL: <https://www.rfc-editor.org/rfc/rfc3552.txt>

17385 **[RFC 3566]**
 17386 Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC
 17387 3552, DOI 10.17487/RFC3552, July 2003.
 17388 URL: <https://www.rfc-editor.org/rfc/rfc3566.txt>

17389 **[RFC 3610]**
 17390 Whiting, D., Housley, R., and N. Ferguson, “Counter with CBC-MAC (CCM)”, IETF RFC 3610, September
 17391 2003.
 17392 URL: <http://www.ietf.org/rfc/rfc3610.txt>

17393 **[RFC 3686]**
 17394 Housley, “Using Advanced Encryption Standard (AES) Counter Mode With IPsec Encapsulating Security
 17395 Payload (ESP)”, IETF RFC 3686, January 2004.
 17396 URL: <http://www.ietf.org/rfc/rfc3686.txt>.

17397 **[RFC 3713]**
 17398 Matsui, et al, “A Description of the Camellia Encryption Algorithm”, IETF RFC 3713, April 2004.
 17399 URL: <http://www.ietf.org/rfc/rfc3713.txt>.

17400 **[RFC 3748]**
 17401 Aboba et al, “Extensible Authentication Protocol (EAP)”, IETF RFC 3748, June 2004.
 17402 URL: <http://www.ietf.org/rfc/rfc3748.txt>.

17403 **[RFC 4269]**
 17404 South Korean Information Security Agency (KISA) “The SEED Encryption Algorithm”, December 2005.
 17405 URL: <https://ftp.rfc-editor.org/in-notes/rfc4269.txt>

17406 **[RFC 4309]**
 17407 Housley, R., “Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security
 17408 Payload (ESP)”, IETF RFC 4309, December 2005.
 17409 URL: <http://www.ietf.org/rfc/rfc4309.txt>

17410 **[RFC 4357]**
17411 V. Popov, I. Kurepkin, S. Leontiev “Additional Cryptographic Algorithms for Use with GOST 28147-89,
17412 GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms”, January 2006.
17413 URL: <http://www.ietf.org/rfc/rfc4357.txt>
17414 **[RFC 4490]**
17415 S. Leontiev, Ed. G. Chudov, Ed. “Using the GOST 28147-89, GOST R 34.11-94, GOST R 34.10-94, and
17416 GOST R 34.10-2001 Algorithms with Cryptographic Message Syntax (CMS)”, May 2006.
17417 URL: <http://www.ietf.org/rfc/rfc4490.txt>
17418 **[RFC 4491]**
17419 S. Leontiev, Ed., D. Shefanovski, Ed., “Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R
17420 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile”, May
17421 2006.
17422 URL: <http://www.ietf.org/rfc/rfc4491.txt>
17423 **[RFC 4493]**
17424 J. Song et al. *RFC 4493: The AES-CMAC Algorithm*. June 2006.
17425 URL: <http://www.ietf.org/rfc/rfc4493.txt>
17426 **[RFC 5705]**
17427 Rescorla, E., “The Keying Material Exporters for Transport Layer Security (TLS)”, RFC 5705, March 2010.
17428 URL: <http://www.ietf.org/rfc/rfc5705.txt>
17429 **[RFC 5869]**
17430 H. Krawczyk, P. Eronen, “HMAC-based Extract-and-Expand Key Derivation Function (HKDF)”, May 2010
17431 URL: <http://www.ietf.org/rfc/rfc5869.txt>
17432 **[RFC 7627]**
17433 K. Bhargavan et al, “Transport Layer Security (TLS) Session Hash and Extended Master Secret
17434 Extension”, IETF RFC 7627, September 2015
17435 URL: <https://www.rfc-editor.org/rfc/rfc7627>
17436 **[RFC 7693]**
17437 M-J. Saarinen, J-P. Aumasson, “The BLAKE2 Cryptographic Hash and Message Authentication Code
17438 (MAC)”, IETF RFC 7693, November 2015
17439 URL: <https://tools.ietf.org/html/rfc7693>
17440 **[RFC 7748]**
17441 Aboba et al, “Elliptic Curves for Security”, IETF RFC 7748, January 2016
17442 URL: <https://tools.ietf.org/html/rfc7748>
17443 **[RFC 8032]**
17444 S. Josefsson, I. Liusvaara, “Edwards-Curve Digital Signature Algorithm (EdDSA)”, IETF RFC 8032,
17445 January 2017
17446 URL: <https://tools.ietf.org/html/rfc8032>
17447 **[RFC 8410]**
17448 S. Josefsson, J. Schaad, “Algorithm Identifiers for Ed25519, Ed448, X25519, and X448 for Use in the
17449 Internet X.509 Public Key Infrastructure”, IETF RFC 8410, August 2018
17450 URL: <https://tools.ietf.org/html/rfc8410>
17451 **[SEC 1]**
17452 Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC) 1:*
17453 *Elliptic Curve Cryptography*. Version 1.0, September 20, 2000.
17454 **[SEC 2]**
17455 Standards for Efficient Cryptography Group (SECG). *Standards for Efficient Cryptography (SEC) 2:*
17456 *Recommended Elliptic Curve Domain Parameters*. Version 1.0, September 20, 2000.
17457

17458 **[WTLS]**
17459 WAP. Wireless Transport Layer Security Version — WAP-261-WTLS-20010406-a. April 2001.
17460 URL: <http://openmobilealliance.org/tech/affiliates/wap/wap-261-wtls-20010406-a.pdf>
17461 **[XEDDSA]**
17462 The XEdDSA and VEdDSA Signature Schemes - Revision 1, 2016-10-20, Trevor Perrin (editor)
17463 URL: <https://signal.org/docs/specifications/xeddsa/>
17464 **[X.500]**
17465 ITU-T. Information Technology — Open Systems Interconnection — The Directory: Overview of
17466 Concepts, Models and Services. February 2001. Identical to ISO/IEC 9594-1
17467 **[X.509]**
17468 ITU-T. Information Technology — Open Systems Interconnection — The Directory: Public-key and
17469 Attribute Certificate Frameworks. March 2000. Identical to ISO/IEC 9594-8
17470

Appendix B. Acknowledgments

B.1 Special Thanks

Substantial contributions to this document from the following individuals are gratefully acknowledged:

Ms. Dina Kurktchi-Nimeh, Oracle

B.2 Participants

The following individuals were members of this Technical Committee during the creation of this document and their contributions are gratefully acknowledged:

Dr. Warren Armstrong, QuintessenceLabs Pty Ltd.

Anthony Berglas, Cryptsoft Pty Ltd.

Mr. Dieter Bong, Utimaco IS GmbH

Hamish Cameron, nCipher

Kenli Chong, QuintessenceLabs Pty Ltd.

Mr. Justin Corlett, Cryptsoft Pty Ltd.

Mr. Tony Cox, TC Logic/Tony Cox

Ms. Valerie Bubba Fenwick, Apple

Ms. Susan Gleeson, Oracle

Tim Hudson, Cryptsoft Pty Ltd. | OpenSSL Software Services Inc.

Mr. Gershon Janssen, Reideate

Mr. Jakub Jelen, Red Hat

Mr. Darren Johnson, THALES

Sun-Ho Lee, MDS Intelligence Inc

John Leiseboer, QuintessenceLabs Pty Ltd.

Mr. John Leser, Oracle

Scott Leubner, THALES

Scott Marshall, Cryptsoft Pty Ltd.

Dr. Michael Markowitz, Information Security Corporation

Mr. Darren Moffat, Oracle

Mr. Tim Ober, THALES

Dr. Florian Poppa, QuintessenceLabs Pty Ltd.

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

Simo Sorce, Red Hat

Mr. Manish Upasani, Utimaco IS GmbH

Ms. Magda Zdunkiewicz, Cryptsoft Pty Ltd.

Appendix C. Revision History

Revision	Date	Editor	Changes Made
WD01	27 September 2023	Greg Scott	Import of PKCS11 V3.1 into Committee Specification -Working Draft 01 - Moving some text to align with updated template formats
WD02	23-October-2023	Greg Scott	Updated from V3.2 Wiki Work Item 14 ; Action Item 06 ; Public Comments Item 4 , Public Comments Item 5
WD03	7-May-2024	Greg Scott, Dieter Bong	Updated from V3.2 Wiki: <ul style="list-style-type: none"> Work Item 1, Work Item 2, Work Item 3, Work Item 4, Work Item 5, Work Item 7, Work Item 8, Work Item 10, Work Item 12 Action Item 8
WD04	10-July-2024	Dieter Bong	Updated from V3.2 Wiki: <ul style="list-style-type: none"> Work Item 6.1 with email comments by D.Bong incorporated, Work Item 6.2, Work Item 6.3, Work Item 6.4, Work Item 11, Work Item 17 Public Comments Item 1 Selected feedback from Robert Künnemann (full stop, ullvFixedBits most significant bits, paragraph break before CKG_GENERATE and CKG_NO_GENERATE, C_Wrapkey without the capital K)
WD05	06-August-2024	Dieter Bong	Updated from V3.2 Wiki: <ul style="list-style-type: none"> Work Item 18, Work item 19 <p>Other updates:</p> <ul style="list-style-type: none"> Fix Mechanisms vs. Functions assign-ment for CKM_SHA3_384_KEY_GEN and CKM_SHA3_512_KEY_GEN References: review, cleanup and consistent notation Addition of note “1” for single-part operations in Table 37, Table 42, Table 43, Table 73 and Table 288 for consistency with text below
WD06	18-October-2024	Dieter Bong	Updated from V3.2 Wiki: <ul style="list-style-type: none"> Work Item 20 Section 6.1.7 as per Action Item 7 <p>Other updates:</p> <ul style="list-style-type: none"> Sections 6.67 ML-DSA, 6.68 ML-KEM, 6.69 SLH-DSA and A.1 Normative References

			<p>updated to comply with final versions of FIPS 203/204/205</p> <ul style="list-style-type: none"> • Section 4.15.3.1 : fixed incorrect integration of FIPS proposal • Table 6, line CKF_DUAL_CRYPT_Operations: reference fixed • Section 6.13.2: fix “decryption” -> “encryption” • Added / updated table names and fixed typos as per discussion “Automated review of our document. OASIS PKCS 11 TC (oasis-open.org)”
WD07	9 December 2024	Dieter Bong	<p>Updated from V3.2 Wiki:</p> <ul style="list-style-type: none"> • Work Item 21 <p>Other Updates:</p> <ul style="list-style-type: none"> • Added reference [RFC 7627] • Bolding unbolded C_*, CKA_*, CKD_*, CKF_*, CKG_*, CKH_*, CKK_*, CKM_*, CKN_*, CKO_*, CKS_*, CKU_*, CKZ_* (except when used in definitions, tables or sample code)
WD08	14 January 2025	Dieter Bong	<p>Updated from V3.2 Wiki:</p> <ul style="list-style-type: none"> • Work Item 22 <p>Other Updates:</p> <ul style="list-style-type: none"> • Section 5.18.7: fix “wrap” -> “unwrap” as per Darren's comment wd07-review-comments-related-to-public-comments-item-1-gcm-ccmwrapivnonceupdate-wd8 • Updates as per Bob's comment draft-07-review... • Bolding unbolded CKR_*, CKT_* (except when used in definitions, tables, sample code and list of return values)
WD09	20 February 2025	Dieter Bong	<p>Updates as per TC Meeting 15 Jan 2025:</p> <ul style="list-style-type: none"> • Section 6.67.3, 6.68.3: make CKA_SEED a “Byte array” and fix footnotes • Section 6.67.4. add CKA_SEED as contributed attribute • Section 6.67.5: remove requirement for C_VerifySignatureInit • Section 4.8.2, 4.15.3.2: replace CKA_VALIDATION_FLAGS by CKA_OBJECT_VALIDATION_FLAGS <p>Other updates:</p>

			<ul style="list-style-type: none"> • Add KEM in section Definitions, fixed references [FIPS 203] and [FIPS 204], updates in sections 6.7, 6.67, 6.68, 6.69 as per Comments by Francois Rousseau • Replace CKT_TRUSTED_DELEGATOR by CKT_TRUST_ANCHOR as per Bob's update in draft-07-review • Change CK_KEY_HANDLE to CK_SP800_108_KEY_HANDLE as per TC Meeting 29 Jan 2025 • Updates as per Review: C_EncapsulateKey versus C_DecapsulateKey OASIS PKCS 11 TC and TC Meeting 12 Feb 2025 • Section 5.21.1 Updated wording as per Clarifications in specification of ML-DSA and SLH-DSA OASIS PKCS 11 TC
WD10	25 February 2025	Dieter Bong	<ul style="list-style-type: none"> • Section 6.67.6: fix hash function used • Sections 6.69.5, 6.69.6: added hash function used • Throughout document: Fix "Byte Array" -> "Byte array" and "Local String" -> "Local string" • Reference [FALCON ROUND3] removed • Section 5.14: fix "for for signing messages"
WD11	10 March 2025	Dieter Bong	<ul style="list-style-type: none"> • Sections 4.1.1 / 4.1.2 / 4.1.2: fixed references to section 5.7 • Added CKR_MECHANISM_PARAM_INVALID to all message-based signing / verification / encryption / decryption functions that have input parameter 'pParameter'; and added CKR_OPERATION_NOT_INITIALIZED to MessageBegin functions for message-based signing / verification / encryption / decryption; as per discussion Extra return values for message based apis • Added CKR_PARAMETER_SET_NOT_SUPPORTED as per discussion New error code for parameter set not supported
WD12	26 March 2025	Dieter Bong	<ul style="list-style-type: none"> • Add C_WrapKeyAuthenticated and C_UnwrapKeyAuthenticated to 3.2 function table
WD13	16 April 2025	Dieter Bong	<ul style="list-style-type: none"> • Removal of editing and formatting instructions • Removal of empty Appendix B • Updated Appendix (former C, now B) Acknowledgements

Appendix D. Notices

Copyright © OASIS Open 2025. 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.